Logical Time in Model-Driven Engineering

Frédéric Mallet

UNIVERSITÉ NICE-SOPHIA ANTIPOLIS

École Doctorale STIC

Habilitation à Diriger des Recherches

Spécialité: **INFORMATIQUE**

par

Frédéric MALLET

Logical Time in Model-Driven Engineering

Soutenue publiquement le 26 novembre 2010 devant le jury composé de:

Rapporteurs	M.	**Robert**	**France**	Professor CSU
	M.	**Suzanne**	**Graf**	DR CNRS/HDR
	M.	**Jean-Pierre**	**Talpin**	DR INRIA/HDR
Président	M.	**François**	**Terrier**	DR CEA/Professor
Examinateurs	M.	**Charles**	**André**	Professor UNS
	M.	**Robert**	**de Simone**	DR INRIA/HDR

Laboratoire I3S, salle de conférences

Preface

In this document, I introduce some work realized over the last few years in collaboration with the team-project Aoste and other people with various backgrounds. AOSTE is a joint team between I3S and INRIA, bi-localized in Sophia Antipolis and Rocquencourt. It builds on previous experience by research scientists from the former TICK and OSTRE INRIA teams, and the I3S SPORTS team from the University of Nice-Sophia antipolis. Aoste tackles several topics in the design methodologies for real-time embedded systems. Here design means altogether:

- High-Level Modeling ;
- Transformation and Analysis ;
- Implementation onto Embedded platforms.

To cover this vast spectrum of subjects we need to specialize the type of formalisms considered. Aoste focuses on synchronous reactive systems, such as Esterel/SyncCharts, and on the AAA methodology. Part of our activity is devoted to enrich the Model Based Design approach with a proper UML modeling diagrams and profiles with elements allowing for efficient modeling and embedding of synchronous designs. The main idea was to extend the Unified Modeling Language (UML) so as it could be used for the design of embedded and real-time systems. UML provides a broad range of diagrams to cover all aspects of a system (*e.g.,* requirements, object models, functional models, state machines, data flows, deployment). It allows extensions through the definition of profiles and seems to be a good front-end to capture models. These models need a precise semantics and depending on the analysis context the same

diagrams may be interpreted with a different semantics. To allow interoperability of models, the expected interpretation must not be left outside the model and should rather be made explicit within the model.

Following some collaborations with the CEA-List and Thalès Research & Technology, we have had the opportunity to contribute to the definition of the UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) by the Object Management Group (OMG). The main part of this contribution was to lead the definition of its model and allocation models. Inspired by our background in synchronous reactive systems, we setup in 2005 to define a language, called the Clock Constraint Specification Language (CCSL), as a support to make explicit the causal and chronological relations between events of a system. Even though it was devised as a companion language for UML/MARTE models, CCSL has been developed independently and is now defined as a model (rather than a language) that can complement not only UML models but also non-UML ones (*e.g.,* ecore-based).

The first part of this document describes CCSL. The remaining two parts give usage examples of CCSL for modeling (Part II) and for verification (Part III).

Acknowledgement

I must say that, as most research works, this work would not have been possible without the help and contribution of several administrative, engineering and research staff and students. So I want to thank every one that felt concerned by this work at some point or another.

I am greatly in debt to the referees and members of my examination committee. In a time where the administrative duties are overwhelming I know how demanding it may be to read and report on dissertations. I am grateful that they took some of their time to do it for me and to be part of my committee.

I also want to thank the Aoste team members, current and paste, for being good sparring partners and our assistants for helping me through the administrative maze. Special thanks to Charles, for his great work in defining CCSL, for his huge contribution to this work and for having always been very helpful and ready to discuss. Many thanks to Robert for, as team leaders should do, providing the resources required to foster research activity (research tracks, fundings and collaboration partners).

I have special thoughts for Fernand who has followed my work since the beginning of my Master. He has been of a great support to decrypt and synthesize the tremendous work of philosophers and physicists who have, for ages, tried to grasp the essential nature of time.

To my parents,
To Iris,
To Luc & Vincent.

Contents

List of Figures

Part I

Logical Time at/as design time

CCSL has arisen from different inspiring models in an attempt to abstract away the data and the algorithms and to focus on events and control. Even though CCSL was initially defined as the time model of the UML profile for MARTE, it has now become a full-fledged domain-specific modeling language for capturing causal, chronological and timed relationships. It is intended to be used as a complement of other syntactic models that capture the data structure, the architecture and the algorithm.

Chapter 1 introduces the background and the historical models of concurrency that have inspired the construction of CCSL. Chapter 2 introduces CCSL and its relationships to the UML and MARTE. Chapter 3 compares CCSL to related concurrent models.

Chapter 1

Background

1.1 Models of Computation and Communication

Embedded systems are built by assembling various kinds of concurrent components : hardware or software. For hardware components, the concurrency is physical because components actually run in parallel; hardware description languages consider components as concurrent processes that communicate through shared signals. Embedded software is increasingly a composition of concurrent processes and departs from traditional software engineering by making explicit the concurrency and by offering mechanisms to model the time. For software components, the concurrency is potential and mainly due to the dependency or rather independence between data (and control) processed by algorithms. In both cases, the components interact in a variety of ways, not limited to the simple transfer of control of the classical synchronous message-passing mechanism predominantly used in software engineering. Components represent computations (or communications) and their interaction are governed by so-called Models of Computation[1] and Communication (MoCC) [Lee00, Jan03]. The heterogeneity of modern systems requires the joint use of several MoCCs and a

[1]Computational models have been developed as early as the 1930s, Turing Machines are examples. The term "Model of Computation" came in use much later in the 1970s. The use of "Model of Computation and Communication" is even more recent (2000s). It underlines the actual effort to design computations and communications separately.

framework to combine them [EJL+03]: state machines for control-dominant aspects, data-flow for data-intensive processing, discrete-event to deal with mainly aperiodic systems, time-triggered or synchronous when predictability is required.

Following Petri's view, computations (processes) are often represented in terms of their events, on which they synchronize, and their effect on local states [2].

Untimed (causal) models focus on the causal dependency and conflicts between events. The word *event* refers either to one occurrence of a certain 'atomic' action or to the set of all the occurrences of this action, depending on the level considered. Atomicity, meaning here instantaneous or without duration, also largely depends on the abstraction level; what is considered as indivisible at a certain level–because its duration is negligible compared to other actions–may have a complex structure when refined. Examples of untimed models include Petri's *Transition Nets* [Pet87], where a transition represents a set of all the occurrences of a given action, *occurrence nets* and *event structures* [NPW81], where each transition represents a single occurrence. It also includes untimed *process algebras* like CCS [Mil80] or CSP [Hoa78], that consider processes as primitive elements and not only events. In that context, event names denote an event *class* and there may be many occurrences of events in a single class. A major difference with causal nets resides in the communications. Communications in causal nets are through events of mutual synchronization, whereas process calculi use (synchronous or asynchronous) channels.

While untimed models mainly deal with *causality*, timed models deal with *sequentialization* [KK98], *i.e.,* an action must occur before another one in some sequentialization, in some temporal ordering of events. Causality clearly induces sequentialization, if a causes b, then b must never be observed before a in any sequentialization. However, causality is not the only reason for sequentialization. There may be some extra-functional reasons: it has been decided that an event should occur periodically every 10 ms because of some (arbitrary or not) reasons, because of the way the sampling mechanism was designed, because by doing so, there will only be some harmonic

[2] Strachey and Scott assumed a view as a function acting in a continuous fashion between datatypes, but the event-based view is closer to our preoccupations.

task sets and some scheduling algorithms would apply. Extra-functional considerations are essential for embedded systems and time is one of the property that must be taken into account, especially in safety-critical hard real-time systems. Time has deliberately been ignored by untimed models, not only to simplify the designs and reasoning but also because it was considered purely extra-functional and so, had to be treated independently of the logical correctness of the design.

"Another detail which we have deliberately chosen to ignore is the exact timing of occurrences of events. The advantage of this is that designs and reasoning about them are simplified, and furthermore can be applied to physical and computing systems of any speed and performance. In cases where timing of responses is critical, these concerns can be treated independently of the logical correctness of the design. Independence of timing has always been a necessary condition to the success of high-level programming languages."

C.A.R. Hoare, Communicating Sequential Processes, 2005.

However, in parallel computations, time is not solely a performance issue as it is in sequential computations, but it can alter the functional behavior. In a system-on-chip with multiple time domains for instance, the relative frequency between the processor clock and the bus clock has an impact on the global functional behavior. In real-time systems, it is sometimes preferable to send approximate data rather than sending them too late (*e.g.,* video processing), this means altering the actual function performed not to miss a deadline. It is sometimes preferable not to send a data when we know it will arrive too late, *e.g.,* to save resources. Time constraints that have an impact on the logical correctness should therefore be integrated into the model. When addressing these problems where time is part of the functional specification, time is qualified as *logical* [Lam78], as opposed to physical. It does not necessarily means that the actual timing must be included in the model but that at least the relative orderings must be considered.

The purpose of this work is to propose a model that combines causal and time aspects.

1.2 Events, causality, exclusion

Let us consider a set E of event occurrences. We do not say what events are, they can be anything performed by a process as time goes on. Two primitive relations are considered between events: *exclusion* and *causal dependency*.

For various reasons, some events may exclude some others. It may be because physically two values cannot be at the same time at some place or because they compete for the same resource. Forwards conflicts in occurrence nets (see Fig. 1.1-a) are simple examples that induce an *exclusion*. a and b shares the same condition, so if a occurs then b will never occur and vice-versa. In, Figure 1.1-b, the relation is directly denoted by the dashed line connecting two events. The upper right figure denotes events as in place/transition nets. The lower right figure is more concise.

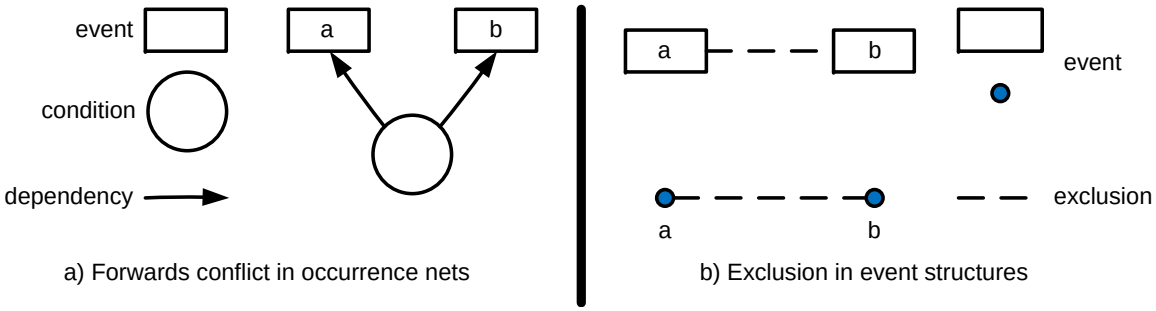

Figure 1.1: Exclusion in occurrence nets and event structures

Some events may cause others. *Causality* is not really the right English word, it rather denotes a necessity [Win08]. In that context, a causes b does not mean that if a occurs then b must occur, but rather that b cannot occur, unless a has already occurred. In the following, we abusively use the terminology *causality* or *causal dependency* to remain consistent with other works on the topic. Figure 1.2-a illustrates the causal dependency on an occurrence net: events a and b cannot occur unless c has already occurred. Figure 1.2-b abstracts away the shared condition and directly represents the causal dependency with a plain arrow directed from the *cause* to the *effect*. It must be acknowledged that this kind of causality is purely logical and not temporal (as might wrongly suggest the present perfect). It does not say anything about the date at which the events occur but only specifies the logical ordering in

which events must occur. It differs from the (temporal) precedence relation introduced later.

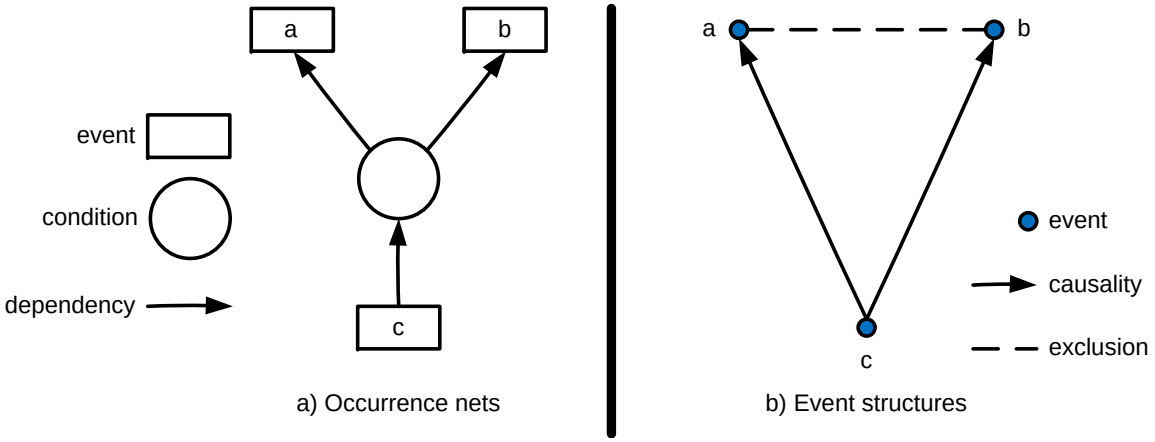

Figure 1.2: Causality and exclusion

There have been many presentations and variants of event structures. In [NPW79], *elementary event structures* are simply a partially ordered set $(E, \leq)$, where E is a set of events (occurrences) and $\leq \subset E \times E$ is a partial ordering over E, called the *causality relation*. Assuming that in physical systems, there is an upper bound on the speed at which causality travels, a property of *finite causes* is considered: $\forall e \in E, \{e' \in E | e' \leq e\}$ is finite. In *event structures*, a *conflict relation* $\# \subset E \times E$, a symmetrical and irreflexive relation on E, is added. The *axiom of conflict heredity* is stated as follows: $\forall e, e', e'' \in E, (e \# e' \wedge e' \leq e'') \Rightarrow e \# e''$.

Later [Win86], the binary relations are replaced by N-ary relations. The *enabling relation* generalizes the causality relation and stands for events that have multiple causes (AND-Causality [Gun92]) (see Fig. 1.3-a). A *consistency predicate*, $Con \subset Fin(E)$ over the finite subsets of E, generalizes the binary conflict relation by picking out some events that can occur together. Figure 1.3-b shows the domain of configurations for a simple consistency predicate. a is not directly in conflict with neither b nor c, since $\{a, b\} \in Con \wedge \{a, c\} \in Con$ but a will never occur when both b and c have occurred. Ultimately, a, b, c cannot all occur: $\{a, b, c\} \notin Con$. The binary exclusion implies the following simple rule on the consistency predicate: $(a \# b) \Rightarrow (\forall X \in Con)(\{a, b\} \not\subseteq X)$.

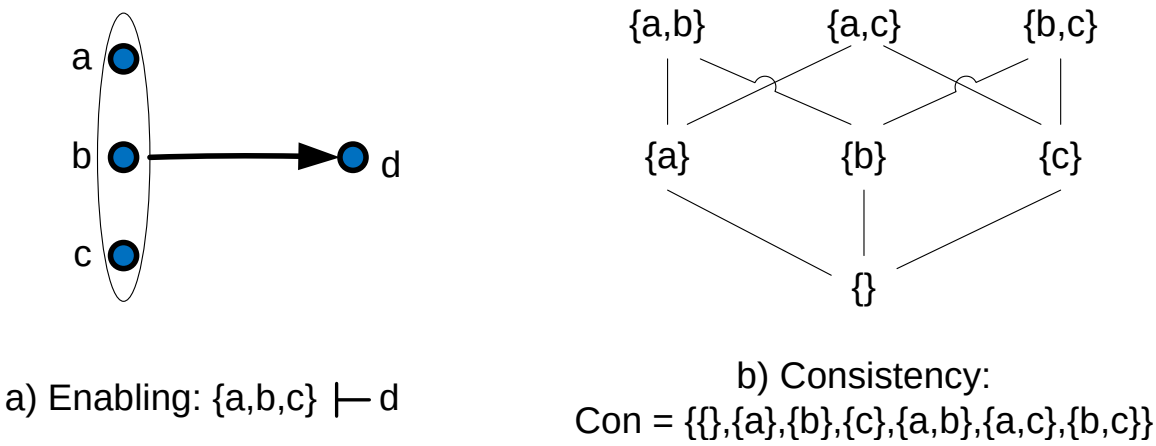

Figure 1.3: Enabling and consistency.

Flow event structures [BC88] are a more flexible notion of event structures obtained by relaxing the axiom of conflict heredity, the finite cause property and by replacing the causal dependency by a weaker (irreflexive) relation, the *flow* relation, denoted $\prec$.

1.3 Tag structures

The notion of *Tag Systems* has been initially introduced in the Lee and Sangiovanni-Vincentelli's (LSV) tagged-signal model [LS98]. LSV is a denotational approach where a system is modeled as a set of behaviors. Behaviors are sets of events. Each event is characterized by a data value and a tag. It departs from the work on event structures by not relying on the category theory to be more accessible. The parallel composition of systems consists in taking the intersection of their corresponding sets of behaviors. Tag Systems gave rise to *Tag Structures* [BCC+08] and to *Tag Machines* [BCCSV05]. Tag structures aim at providing a compositional theory of heterogeneous reactive systems. They restrain the generality of tag systems and give more structure to the behaviors, which become finite sets of signals, signals being *sequences* of events. They also introduce the concept of *stretching* to allow certain deformations of the tags for behaviors. The stretching mechanism allows an entire

characterization of the MoCCs. Parallel composition is allowed through the *fibered product* of tag structures and requires an appropriate algebra. More formally, a tag structure is a triple $\langle \mathcal{T}, \leq, \Phi \rangle$, where:

- $\mathcal{T}$ is a set of tags and $(\mathcal{T}, \leq)$ is a partial order;
- Φ is a set of increasing total functions $\phi : \mathcal{T} \to \mathcal{T}$, the set of stretching functions.

This work departs from the event structures by considering timed systems. Tags are time stamps. For instance, the tag structure $\langle \mathbb{N}, \leq, \{id\} \rangle$, where $\leq$ is the usual total order on natural numbers, can be used in modeling synchrony. Tags indicate the reaction indices. The tag structure $\langle \mathbb{R}_+, \leq, \{id\} \rangle$ can be used in modeling time-triggered systems. Tags become real-time dates.

The stretching function Φ allows deformations of the tags. When $\Phi = \{id\}$ no deformations are allowed and the structure is rigid. When, for instance, Φ is the set of all dilating increasing functions, *i.e.,* functions ϕ such that $\phi(\tau) \geq \tau$ for all $\tau \in \mathcal{T}$, then the tags represent the earliest possible dates for execution of events.

Tag structures then consider a set of variables $\mathcal{V}$. A behavior σ considers a finite subset $V \subset \mathcal{V}$ with domain D and is a mapping defined as follows:

$$\sigma \in V \to \mathbb{N} \to (\mathcal{T} \times D)$$

meaning that, for each $v \in V$, the n^{th} occurrence of v in behavior σ has tag $\tau \in \mathcal{T}$ and value $x \in D$. For a given variable and a given behavior the *clock* is extracted. The clock of v in σ is the first projection of the map $\sigma(v)$ and must be an increasing (order-preserving) function $f : \mathbb{N} \to \mathcal{T}$, *i.e.,* $f(\leq_{\mathbb{N}}) \subseteq \leq_{\mathcal{T}}$.

Figure 1.4 shows an example, where values are not displayed. $\mathcal{T}$ is a set of tags and the solid arrows denote the partial order $\leq$. Each bold line represents a behavior for a given variable. The dashed lines are the clocks for this behavior, the blue (upper) ones for variable $v1$ and the red (lower) ones for variable $v2$. Note that, the third event of $v2$ in σ has the same tag than the second event of $v1$ in σ. The two events occur at the same time, they are *coincident*.

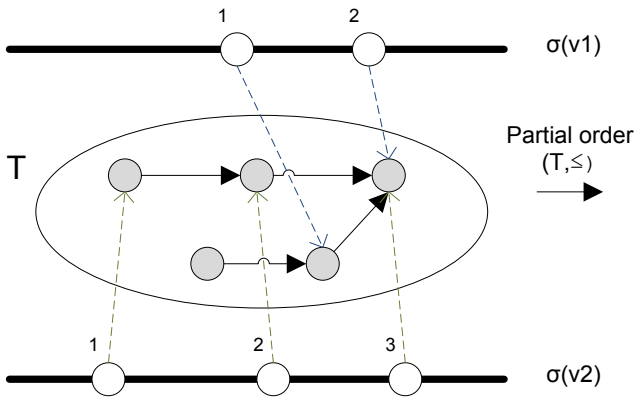

Figure 1.4: Tag structure by example

We consider that *clocks* are central in such a model as they are in synchronous languages. We propose a time model that abstracts away the values and focuses on the clocks themselves. Figure 1.5 shows the same example with logical clocks. The clock $c1$ represents the clock of $\sigma(v1)$, whereas $c2$ represents the clock of $\sigma(v2)$. Clock c is introduced to capture the lonely tag not used in any of these two behaviors. The plain vertical connection (in red) denotes a coincidence. The two instants (circles) represent occurrences of events that are *a priori* independent. Forcing a simultaneous occurrence of the events is done by mapping them to the same tag in the tag structure model and by adding a *coincidence* relation in our model. The dashed arrow denotes the precedence relationship that gives the ordering between the occurrences of the two events. In this model, our clock is more an element of $2^{\mathcal{T}}$ with a proper total order relation rather than a mapping to $\mathcal{T}$ ($\mathcal{T}^{\mathbb{N}}$) as in tag structures.

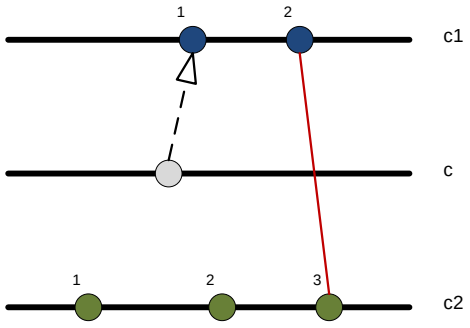

Figure 1.5: Logical clocks by example

The next section discusses further the notions of logical time and clocks independently of any specific model.

1.4 Logical time and logical clocks

Tag structures go beyond event structures by considering tags, which provide a support to model timed systems. The word *time* is used here in a very broad sense where the relative ordering between events primes over the actual date. This *logical time* is widely used in models dedicated to embedded, real-time, reactive systems and in particular it is the time model in use in synchronous languages [BCE+03]. Discrete-time *logical clocks* (or simply logical clocks) represent/measure the logical time. Clocks that refer to the "physical time" or real-time are called *chronometric clocks*. Logical clocks as originally defined by Lamport [Lam78] are a special case introduced as a support to build a distributed synchronization algorithm.

Logical time is sometimes qualified as multiform. Indeed, when renouncing to represent the physical time only, the same model can be used to represent chronological relationships between events on different natures. Chronology being the order in which a series of events occur. The events can be ticks of a processor or cycles of a communication bus, but also ticks emit by sensor that measures the rotation degree of a crankshaft in an engine or the sending of a message between two components.

(Logical) clocks are central in synchronous languages and play an important role in tag structures. We propose a model called the Clock Constraint Specification Language (CCSL) dedicated to building logical clocks and expressing relations on clocks. CCSL is described in the following chapter.

Chapter 2

The Clock Constraint Specification Language

The Clock Constraint Specification Language (CCSL) has initially been introduced as a companion language for the UML profile for MARTE (Modeling and Analysis of Real-Time and Embedded systems). It has then become a domain-specific language (DSL) on its own and it is now developed independently. It aims at being a specification language to equip syntactic models (UML-based or other DSLs) with a timed causality model that explicitly expresses the causal and chronological relationships amongst events of the modeled systems.

The initial intent of MARTE being to cover both design and analysis, a large set of CCSL constraints have been introduced for convenience on top of a relatively small set of kernel primitives. Section 2.1 introduces the CCSL time model. Section 2.2 discusses some fundamental coincidence and precedence constraints. Then, Section 2.3 presents the integration of this model into UML, through MARTE stereotypes.

2.1 Instants, clocks and time structures

CCSL focuses on the events and their occurrences and abstracts away the values. This is a major difference with other related models (Signal, Tag structures), which usually

consider an event as a pair value/time stamp. In CCSL, the value may be added as an annotation and never influences the scheduling in any way. CCSL deals with sets of event occurrences $\mathcal{I}$, called instants to avoid the possible confusion between the events and their occurrences. On these events we build a *time structure* $\langle \mathcal{I}, \prec, \equiv \rangle$ by considering the two primitive binary relations: *strict precedence* (denoted $\prec$) and the *coincidence* (denoted $\equiv$). $\prec$ is irreflexive and transitive, it is only a partial relation and therefore is not asymmetric, there can be instants that are not related to each other. The coincidence is a direct emanation of synchronous languages. $\equiv$ is a partial equivalence relation, *i.e.,* reflexive, transitive and symmetric. The coincidence is the major difference with the general net theory and the event structures because it forces one instant to occur when another has occurred or prevents an event from occurring when another one is not ready. The coincidence is universal, no referential can tell apart two coincident instants. Petri's model (following the relativity theory) restricts coincidence to single points in space-time. In CCSL, the coincidence relationship "melts" *a priori* independent points (instants) to reflect design choices and thus is a foundational relation.

From these two relations, we build three more: *exclusion* (denoted $\#$), *precedence* (denoted $\preccurlyeq$) and *independence* (denoted $\|$). The exclusion is when the instants can never be coincident. Note, that this definition of exclusion is weaker than the exclusion of event structures (see section 1.2), since it does permanently prevent the other event from occurring but just prevents them from being coincident. The precedence is the union of the strict precedence and the coincidence. It is the equivalent of the *causality relation* in event structures and $\preccurlyeq$ is a partial pre-order, *i.e.,* it is reflexive and transitive but neither antisymmetric, nor asymmetric. If $a \preccurlyeq b$ and $b \preccurlyeq a$ then a is not necessarily identical to b but is *coincident* with it ($a \equiv b$). The graphical representation of instant relations is given in Figure 2.1.

All these relations are defined on instants (event occurrences). However, in a specification, it is more likely to talk about the events themselves and therefore we define a *clock* c as a totally ordered set of instants $\mathcal{I}_c$. $\langle \mathcal{I}_c, \preccurlyeq_c \rangle$ is a total order[1]. $\preccurlyeq_c$ is

[1]Colors distinguish the total order on clocks in blue from the partial order on the time structure in red

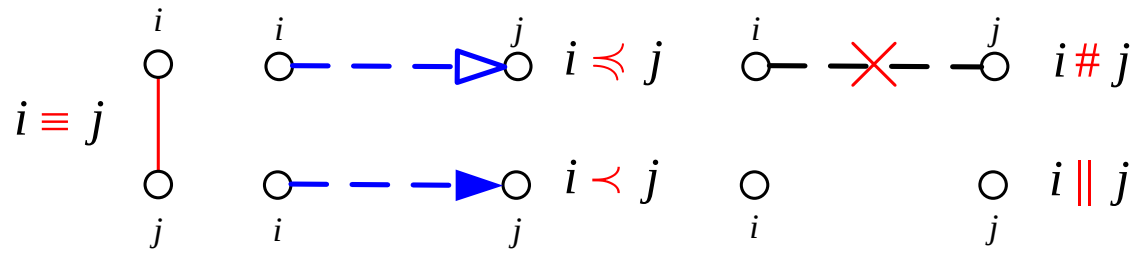

Figure 2.1: Graphical representation of instant relations

antisymmetric since no two instants of a given clock can be coincident without being identical. We chose the word clock, instead of events or signals to avoid confusion with these highly overridden words. The word comes from the synchronous languages, in which the clock is a pure signal (without values). It represents events, but the word event is sometimes used to denote the occurrences and sometimes to denote the classes of occurrences. Additionally, event also exists in UML and we therefore needed another word. Note that the set of instants can be either dense or discrete. A *discrete-time clock* is a clock c with a discrete set of instants $\mathcal{I}_c$. Since $\mathcal{I}_c$ is discrete, it can be indexed by natural numbers in a fashion that respects the ordering on $\mathcal{I}_c$: let $\mathbb{N}^\star = \mathbb{N} \setminus \{0\}$, $\text{idx} : \mathcal{I}_c \to \mathbb{N}^\star$, $\forall i \in \mathcal{I}_c$, $\text{idx}(i) = k$ if and only if i is the k^{th} instant in $\mathcal{I}_c$. $c[k]$ is defined so as $\forall k \in \mathbb{N}^\star, \text{idx}(c[k]) = k$. $^\circ i$ is the unique immediate predecessor of i in $\mathcal{I}_c$. For simplicity, we assume a virtual instant (called birth) the index of which is 0, and which is considered as the immediate predecessor of the first instant.

From a set of clocks C, we build a time structure $\langle \mathcal{I}, \prec, \equiv \rangle$ such that

- $\mathcal{I} = \bigcup_{c \in C} \mathcal{I}_c$
- $\prec = \bigcup_{c \in C} \prec_c$, where $\prec_c$ is the reflexive reduction of $\preccurlyeq_c$ for a clock c
- $\equiv = Id_{\mathcal{I}}$

Instant relations are then extended to clock relations, which usually represent infinitely many instant relations at once. The following section explains how, given a set of clocks and its underlying time structure, we augment it by considering a given specification, a set of clock relations Rel.

2.2 Clock constraints

Specifying a full time structure using only instant relations is not realistic, all the more so since a clock usually has an infinite number of instants therefore forbidding an enumerative specification of instant constraints. Instead of defining individual instant pairings, a clock constraint specifies generic associations between (infinitely) many instants of the constrained clocks.

In this section we define the most general clock constraints and we introduce some usual constraints, derived from the basic ones. The clock constraints are classified in three main categories: 1) coincidence-based constraints, 2) precedence-based constraints, and 3) mixed constraints.

Actually, most constraints only partially constrain the systems and therefore several time structures are possible. So a specification induces a set of time structures that share the same set of instants $\mathcal{I}$. For one given time structure TS, $\mathcal{I}_{TS} = \mathcal{I}$ denotes its set of instants, $\prec_{TS}$ is its precedence relation and $\equiv_{TS}$ its coincidence relation.

2.2.1 Coincidence-based clock constraints

Coincidence-based clock constraints are classical in synchronous languages and can then be very easily specified with such languages.

Sub-Clocking is the most basic coincidence-based clock constraint relationship. Let a, b be two clocks. The clock relation b isSubClockOf a imposes b to be a sub-clock of a. Intuitively, this means that each instant in b is coincident with exactly one instant in a (Figure 2.2). More formally, a time structure $TS = \langle \mathcal{I}, \prec_{TS}, \equiv_{TS} \rangle$ satisfies the clock relation b isSubClockOf a if and only if there exists an injective

mapping $h : \mathcal{I}_b \to \mathcal{I}_a$ such that :

(1) h is order preserving:

$$(\forall i, j \in \mathcal{I}_b)\ (i \preccurlyeq_b j) \implies (h(i) \preccurlyeq_a h(j))$$

(2) an instant of $\mathcal{I}_b$ and its image are coincident:

$$(\forall i \in \mathcal{I}_b)\ i \equiv_{TS} h(i)$$

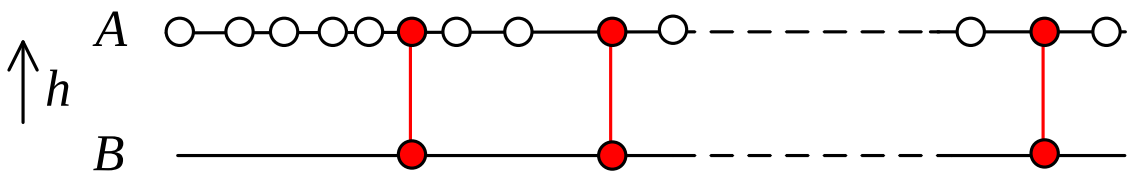

Figure 2.2: Coincidence-based clock constraint

In what follows, this constraint is denoted as $b \boxed{\subset} a$, read "b is a sub-clock of a" or equivalently "a is a super-clock of b".

2.2.2 Derived coincidence-based clock constraints

h can be specified in many different ways.

Equality $a \boxed{=} b$ is the symmetric relation that makes the two clocks a and b "synchronous": h is a bijection and the instants of the two clocks are pair-wise coincident. It is strictly equivalent to $b \boxed{=} a$.

Other coincidence-based *clock expressions* allow the creation of a new clock, sub-clock of a given clock (denoted $new\ clock \triangleq defining\ expression$). Three such clock expressions are presented hereafter.

Restriction $b \triangleq a$ restrictedTo P where a is a given clock, b is a new clock, and P is a predicate on $\mathcal{I}_a \times \mathcal{I}_b$, such that

$$(\forall i \in \mathcal{I}_A, \forall j \in \mathcal{I}_B)\ i \equiv h(j) \iff P(i,j) = \mathsf{true}$$

Filtering $b \triangleq a$ filteredBy w, where a and b are discrete-time clocks, and w is a binary word. For filtering, the associated predicate is such that

$$(\forall i \in \mathcal{I}_a, \forall j \in \mathcal{I}_b)\big(P(i,j) = \mathsf{true} \iff \mathrm{idx}_a(i) = w \uparrow \mathrm{idx}_b(j)\big)$$

where $w \uparrow k$ is the index of the k^{th} 1 in w. The use of infinite k-periodic binary words in this kind of context has previously been studied in N-Synchronous Kahn networks [CDE+06]. This constraint is frequently used in clock constraint specifications and is denoted $A \blacktriangledown w$ in this document. It allows the selection of a subset of instants, on which other constraints can then be enforced.

In what follows, a (periodic) binary word is denoted as $w = u(v)^\omega$, where u is called the *transient* part of w and v its *periodic* part. The power ω means that the periodic part is repeated an unbounded number of times. So, $u(v)^\omega$ denotes $u.v.v.\cdots.v.\cdots$.

Periodicity Defining the periodicity of discrete clocks consists in using a binary word with a single 1 in the periodic part. b isPeriodicOn a period p offset d defines a periodic clock b. The same clock can be built with a filtering $b \triangleq a \blacktriangledown 0^d.(1.0^{p-1})^\omega$. In this expression, for any bit x, x^0 stands for the empty binary word. Note that this is a very general definition of periodicity that does not require a to be chronometric contrary to the usual definition.

2.2.3 Precedence-based clock constraints

Precedence-based clock constraints are easy to specify with concurrent models like Petri nets but are not usual in synchronous languages. A discussion on main differences with Time Petri nets can be found in some of our previous work [MA09].

Precedence The clock constraint Precedence consists in applying infinitely many precedence instant relations. Two forms can be distinguished: the strict precedence a strictly precedes b, and the non strict precedence a precedes b. Intuitively, this means that each instant in b follows one instant in a (Fig. 2.3). More formally, a time structure $TS = \langle \mathcal{I}, \prec_{TS}, \equiv_{TS} \rangle$ satisfies the clock relation a precedes b if and only if there exists an injective mapping $h : \mathcal{I}_b \to \mathcal{I}_a$ such that :

(1) h is order preserving:

$$(\forall i, j \in \mathcal{I}_b)\ (i \preccurlyeq_b j) \implies (h(i) \preccurlyeq_a h(j))$$

(2) an instant of $\mathcal{I}_b$ and its image are ordered:

$(\forall i \in \mathcal{I}_b)\big(h(i) \preccurlyeq_{TS} i\big)$ for the non strict precedence

$(\forall i \in \mathcal{I}_b)\big(h(i) \prec_{TS} i\big)$ for the strict precedence

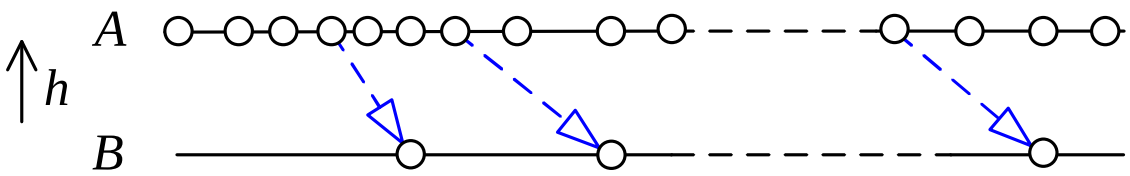

Figure 2.3: Precedence-based clock constraint

2.2.4 Derived precedence-based clock constraints

When a and b are discrete clocks, the precedence relationship gives rise to more specific constraints. Three often used precedence constraints are discussed here.

Discrete precedence A time structure TS satisfies a strictly precedes b (denoted $a \boxed{\prec} b$) iff

$$(\forall i \in \mathcal{I}_b)(k = \mathrm{idx}_b(i)) \implies a[k] \prec_{TS} b[k]$$

There also exists a weak form (denoted $a \boxed{\preccurlyeq} b$) of this constraint where $\prec_{TS}$ is replaced by $\preccurlyeq_{TS}$.

Alternation A time structure TS satisfies a alternatesWith b (denoted $a \boxed{\sim} b$) iff:

$$\left(a \boxed{\prec} b\right) \wedge \left(b \boxed{\prec} a'\right), \text{where } a' \triangleq a \blacktriangledown 0.1^{\omega}$$

The equivalent following specification uses instant relations instead of clock relations, $(\forall i \in \mathcal{I}_a)(k = \text{idx}_a(i)) \implies (a[k] \prec_{TS} b[k] \wedge b[k] \prec a[k+1])$.

Synchronization A time structure TS satisfies a synchronizesWith b (denoted $a \boxed{\bowtie} b$) iff

$$\left(a \boxed{\prec} b'\right) \wedge \left(b \boxed{\prec} a'\right)$$
$$\text{where } a' \triangleq a \blacktriangledown 0.1^{\omega}, \text{and } b' \triangleq b \blacktriangledown 0.1^{\omega}$$

This constraint can also be expressed using instant relations:

$$(\forall k \in \mathbb{N}^{\star})(a[k] \preccurlyeq_{TS} b[k+1]) \wedge b[k] \preccurlyeq_{TS} A[k+1]).$$

Precedences used in the definition of *Alternation* and *Synchronization* can be non-strict precedences, thus there exist four different variants of these clock relations. Another extension considers instants by "packets". For instance, a by α strictly precedes b by β (denoted $a/\alpha \boxed{\prec} b/\beta$) is a short notation for

$$\left(a_f \boxed{\prec} b_s\right)$$
$$\text{where } a_f \triangleq a \blacktriangledown \left(0^{\alpha-1}.1\right)^{\omega}, \text{and } b_s \triangleq b \blacktriangledown \left(1.0^{\beta-1}\right)^{\omega}$$

2.2.5 Mixed constraints

Mixed constraints combine coincidences and precedences. They are used to synchronize clock domains in globally asynchronous and locally synchronous models.

Sampling The commonest constraint of this kind is the *Sampling* constraint. $c \triangleq a$ sampledOn b, where b and c are discrete clocks and a can be either discrete or dense, constrains the clocks a, b and c so that c is a sub-clock of b that ticks only after a tick of a (Fig. 2.4). A time structure TS satisfies $c \triangleq a$ sampledOn b iff

$$(\forall i_c \in \mathcal{I}_c)(\exists i_b \in \mathcal{I}_b)(\exists i_a \in \mathcal{I}_a)(i_c \equiv_{TS} i_b) \wedge (i_a \preccurlyeq_{TS} i_b) \wedge (^\circ i_b \prec_{TS} i_a)$$

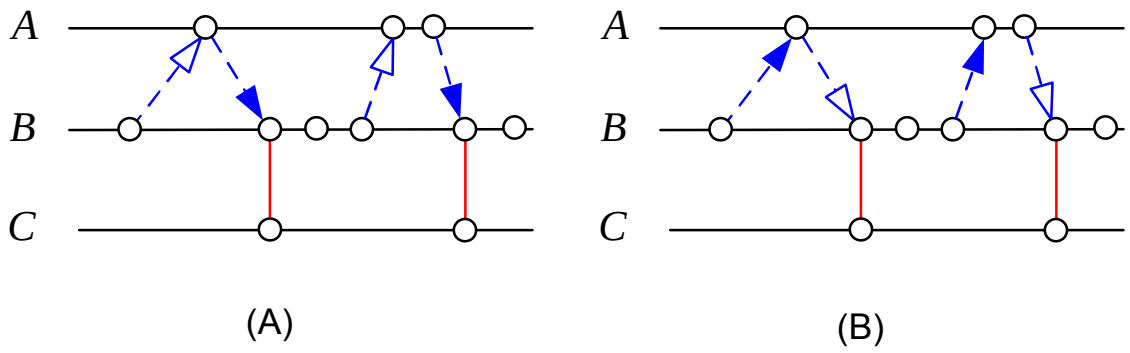

Figure 2.4: Sampling constraints

A time structure TS satisfies the strict form $c \triangleq a$ strictly sampledOn b iff:

$$(\forall i_c \in \mathcal{I}_c)(\exists i_b \in \mathcal{I}_b)(\exists i_a \in \mathcal{I}_a)(i_c \equiv_{TS} i_b) \wedge (i_a \prec_{TS} i_b) \wedge (^\circ i_b \preccurlyeq_{TS} i_a)$$

Delay A slight variation of the sampling is the *Delay* constraint. $c \triangleq a$ delayedFor N b (also denoted $c \triangleq a \ \$_b \ N$) samples the clock a on the N^{th} occurrence of the discrete clock b. c is necessary discrete and is a subclock of b. a can be either discrete or dense.

Note that this operator is polychronous, contrary to usual synchronous delay operators (pre in Lustre, \$ in Signal). There is also a binary variant of this ternary operator denoted \$ and which is equivalent to a delay when a is synchronous with b. $c \triangleq a \ \$ \ N$ is equivalent to $c \triangleq a \ \$_a \ N$.

2.2.6 TimeSquare

Most constraints only partially constrain the system, which means that several (possibly infinitely many) time structures may satisfy a set of constraints. TimeSquare has been built to implement CCSL and provide support for the analysis of CCSL specifications. Several kinds of analyses are possible. Given a set of clocks and a set of constraints, the clock calculus engine can find one possible time structure that satisfies all the constraints. Actually, this is done in simulation and the time structure is only built partially up to the current simulation point. Since it is not always possible to choose a single time structure in a deterministic way the simulation may be non determinate. The clock calculus relies on CCSL structural operational semantics (SOS) to run the simulation. A description of the SOS semantics is available in [And09]. The CCSL constraints are encoded as a set of Boolean equations that determines at each step, which clock can is enabled and which one is not. With non deterministic specifications several solutions (each of which may involve several clocks) may be valid. A simulation policy is then used to pick one solution and fire the associated clocks. Depending on which clocks were fired, the rewriting rules allows to process a new set of boolean equations. This ends the current simulation step and the simulation engine proceeds to the next one. Several simulation policies are offered. For instances, the *random policy* randomly chooses one possible solution amongst the set of solutions. The *minimum policy* chooses a consistent solution where the number of firing clocks is minimal. The *maximum policy* chooses a consistent solution where the number of firing clocks is maximal.

Another kind of analyses is also possible. Given a time structure (*e.g.,* defined as a by-product of the execution of some code) and a CCSL specification, one can verify whether the time structure satisfies all the constraints in the specification. This aspect is described further in the third part of this document. The time structure is given by an existing implementation of the system on which we have no control. The implementation forces some clocks to tick while preventing others to be fired. Our observer-based verification technique can be used to check that the implementation satisfies all the constraints of the specification.

TimeSquare has been implemented as a set of Eclipse plugins and is available for

download at `http://www.inria.fr/sophia/teams/aoste/dev/time_square`. TimeSquare is protected by the APP (Agence pour la protection des programmes).

2.3 UML for Real-Time and Embedded systems

2.3.1 UML, SPT, Marte

The Unified Modeling Language (UML) [OMG09b] is a general-purpose modeling language specified by the Object Management Group (OMG). It proposes graphical notations to represent all aspects of a system from the early requirements to the deployment of software components, including design and analysis phases, structural and behavioral aspects. As a general-purpose language, it does not focus on a specific domain and maintains a weak, informal semantics to widen its application field. However, when targeting a specific application domain and especially when building trustworthy software components or for critical systems where life may be at stake, it is absolutely required to extend the UML and attach a formal semantics to its model elements. The simplest and most efficient extension mechanism provided by the UML is through the definition of profiles. A UML profile adapts the UML to a specific domain by adding new concepts, modifying existing ones and defining a new visual representation for others. Each modification is done through the definition of annotations (called stereotypes) that introduce domain-specific terminology and provide additional semantics. However, the semantics of stereotypes must be compatible with the original semantics (if any) of the modified or extended concept.

The UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE [OMG09a]) extends the UML with concepts related to the domain of real-time and embedded systems. It supersedes the UML profile for Schedulability, Performance and Time (SPT [OMG05]) that was extending the UML 1.x and that had limited capabilities. MARTE has three parts: *Foundations*, *Design* and *Analysis*.

The foundation part is itself divided into five chapters: *CoreElements*, *NFP*, *Time*, *Generic Resource Modeling* and *Allocation*. *CoreElements* defines configurations and modes, which are key parameters for analysis. In real-time systems, preserving the

non-functional (or extra-functional) properties (power consumption, area, financial cost, time budget...) is often as important as preserving the functional ones. The UML proposes no mechanism at all to deal with non-functional properties and relies on mere String for that purpose. NFP (Non Functional Properties) offers mechanisms to describe the quantitative as well as the qualitative aspects of properties and to attach a unit and a dimension to quantities. It defines a set of predefined quantities, units and dimensions and supports customization. NFP comes with a companion language called VSL (Value Specification Language) that defines the concrete syntax to be used in expressions of non-functional properties. VSL also recommends syntax for user-defined properties. Time is often considered as an extra-functional property that comes as a mere annotation after the design. These annotations are fed into analysis tools that check the conformity without any actual impact on the functional model: *e.g.,* whether a deadline is met, whether the end-to-end latency is within the expected range. Sometimes though, time can also be of a functional nature and has a direct impact on what is done and not only when it is done. All these aspects are addressed in the time chapter of MARTE [AMdS07]. The next section elaborates on the time profile.

The design part has four chapters: High Level application modeling, Generic component modeling, Software Resource Modeling, and Hardware Resource Modeling. The first chapter describes real-time units and active objects. Active objects depart from passive ones by their ability to send spontaneous messages or signals, and react to event occurrences. Passive objects can only answer to the messages they receive. The three other parts provide a support to describe resources used and in particular the execution platform on which the application may run. A generic description of resources is provided, including stereotypes to describe communication media, storages and computing resources. Then this generic model is refined to describe software and hardware resources along with their non-functional properties.

The analysis part also has a chapter that defines generic elements to perform model-driven analysis on real-time and embedded systems. This generic chapter is specialized to address schedulability analysis and performance analysis. The chapter on schedulability analysis is not specific to a given technique and addresses various

formalisms like the classic and generalized Rate Monotonic Analysis (RMA), holistic techniques, or extended timed automata. This chapter provides all the keywords usually required for such analyses. In Chapter 5 of this document, we follow a rather different approach and instead of focusing on syntactic elements usually required to perform schedulability analysis (periodicity, task, scheduler, deadline, latency), we show how we can use MARTE time model and its companion language CCSL to build libraries of constraints that reflect the exact same concepts. Finally, the chapter on performance analysis, even if somewhat independent of a specific analysis technique, emphasizes on concepts supported by the queueing theory and its extensions.

MARTE extends the UML for real-time and embedded systems but should be refined by more specific profiles to address specific domains (avionics, automotive, silicon) or specific analysis techniques (simulation, schedulability, static analysis). The three examples addressed here consider different domains and/or different analysis techniques to motivate the demand for a fairly general time model that has justified the creation of MARTE time subprofile.

2.3.2 Marte Time profile

Time in SPT is a *metric* time with implicit reference to physical time. As a successor of SPT, MARTE supports this model of time. UML 2, issued after SPT, has introduced a model of time called *SimpleTime* [OMG09b, Chap. 13]. This model also makes implicit reference to physical time, but is too simple for use in real-time applications and was, right from the beginning, expected to be extended in dedicated profiles.

MARTE goes beyond SPT and UML 2. It adopts a more general time model suitable for system design. In MARTE, Time can be *physical*, and considered as *dense*, but it can also be *logical*, and related to user-defined clocks. Time may even be *multiform*, allowing different times to progress in a non-uniform fashion, and possibly independently to any (direct) reference to physical time.

In MARTE, time is represented by a collection of *Clocks*. Each clock specifies a totally ordered set of instants (see Section 2.1). There may be dependence relationships between instants of different clocks. In practice, clocks should be associated

with events whose occurrence should be constrained or with events used to constrain others. In UML, TimeEvent is specialization of Event stating that this special kind of events can carry a time stamp. Our intent is to allow any kind of event to carry a time stamp. For instance, the goal would be to constrain the way a message is sent or received, a behavior starts or finishes its execution. Therefore time (or causality) must be orthogonal to the various events and not a special kind of events.

The MARTE Time profile defines two stereotypes ClockType and Clock to represent the concept of clock. ClockType gathers common features shared by a family of clocks. The ClockType fixes the nature of time (dense or discrete), says whether the represented time is linked to physical time or not (respectively identified as chronometric clocks and logical clocks), chooses the type of the time units. A Clock, whose type must be a ClockType, carries more specific information such as its actual unit, and values of quantitative (resolution, offset, etc.) or qualitative (time standard) properties, if relevant.

TimedElement is another stereotype introduced in MARTE. A timed element is *explicitly bound* to at least one clock, and thus closely related to the time model. For instance, a TimedEvent, which is a specialization of TimedElement extending UML Event, has a special semantics compared to usual events: it can occur only at instants of the associated clock. In a similar way, a TimedValueSpecification, which extends UML ValueSpecification, is the specification of a set of *time values* with explicit references to a clock, and taking the clock's unit as time unit. Thus, in a MARTE model of a system, the stereotype TimedElement or one of its specializations is applied to model elements which have an influence on the specification of the temporal behavior of this system.

The MARTE Time subprofile also provides a model library named TimeLibrary. This model library defines the enumeration TimeUnitKind which is the standard type of time units for chronometric clocks. This enumeration contains units like s (second), its submultiples, and other related units (minute, hour...). The library also predefines a clock type (IdealClock) and a clock (idealClk) whose type is IdealClock. idealClk is a dense chronometric clock with the second as time unit. This clock is assumed to be an ideal clock, perfectly reflecting the evolutions of physical time. idealClk should be

imported in user's models with references to physical time concepts (*i.e.,* frequency, physical duration, etc.).

2.3.3 UML and formal methods

The UML is a general-purpose modeling language. Its semantics is described in English in the OMG specification. Several aspects are left unspecified in so-called *semantic variation points* to allow user-defined adaptations for addressing domain-specific issues. There have been many efforts to use the UML in formal environments to address critical systems or domains where formal verification is mandatory. A brief classification is attempted hereafter even though it is usually acknowledged [BLMF00] that building a full taxonomy is difficult:

1. The first kind consists in using a formal language to build UML expressions and/or constraints. Since, UML does not enforce any specific syntax for expressions and constraints, any language can be used. Even though the Object Constraint Language (OCL) [OMG06] may seem the most natural choice, formal languages (*e.g.,* Z [GL00], Labelled Transition Systems [SCK09]) are also used to specify invariants in UML use cases or pre-condidions/post-assertions on UML operations. Then, scenarios (UML interaction diagrams) or behaviors (state machines or activities) are statically analyzed [CPC+04, GBR07, YFR08] to check whether the use case invariants and the post-assertions hold.

2. The second kind of approaches is *transformational*. It consists in transforming every UML model element into a model element of a formal language (for instances, Petri Nets [Stö05] or π-calculus [YsZ03]). After transformation, various analyses become possible, like symbolic model-checking [Esh06].

3. In the third kind, the semantics resides in annotations (stereotypes). For instance, the semantics of Concurrent Sequential Processes [Hoa78] can be given to a UML state machine provided that a UML profile for Concurrent Sequential Processes [FGL+08] is defined.

4. In the fourth kind, only a subset of the UML is allowed to make it suitable for a specific domain, *e.g.,* the UML state machines can be reduced to timed automata [AD94] thus giving access to a whole family of formal tools [BDL+06].

In all cases, the semantics remains outside the model and therefore the model may be interpreted differently by another tool or another user. The purpose of CCSL is to provide a domain-specific language (DSL) to build an explicit semantic model that would be linked into the model itself to define its time/causal semantics. Our approach is not specific to UML models and can be used with domain-specific models as well. We propose to use the MARTE time model and its companion language CCSL (Clock Constraint Specification Language) to build the semantic model.

2.3.4 UML and synchronous languages

Synchronous languages [Hal92, BCE+03] are well-suited to formal specification and analysis of reactive system behavior. They are even more relevant with safety-critical applications where lives may be at stake. However, to cover a complete design flow from system-level specification to implementation, synchronous languages need to interoperate with other, more general, specification languages. One of the candidates is the UML associated with SysML, the UML profile for systems engineering [Wei08, OMG08]. This is very tempting since synchronous languages internal formats rely on state machines or data flow diagrams both very close to UML state machines and activities. Moreover, SyncCharts [And96] are a synchronous, formally well-founded, extension of UML state machines and are mathematically equivalent to Esterel [Ber00], one of the three major synchronous languages. As for SysML, it adds two constructs most important for specification: requirements and constraint blocks (see 2.3.5).

There have been attempts to bridge the gap between the UML and synchronous languages. Some [LD06] choose to import UML diagrams into Scade, a synchronous environment that combines *Safe State Machines* (SSM—a restriction of SyncCharts) together with block diagrams, the semantics of which is based on Lustre. Others [BRG+01] prefer to define an operational semantics of UML constructs with a synchronous language, like SIGNAL. In both cases, the semantics remains outside

the UML and within proprietary tools. Other tools, from the same domain, would interpret the same models with a completely different semantics, not necessarily compatible. Therefore, it is impossible to exchange diagrams between tools, not only because of syntactical matters but also for semantic reasons. Different environments are then competing rather than being complementary. To provide full interoperability between tools of the embedded domain, the UML absolutely requires a timed causality model. The UML Profile for MARTE has introduced a time model with that purpose (see 2.3.1). Its companion language, CCSL is advertised as a pivot language to make explicit the interactions between different models of computations [MAdS08], like Ptolemy directors [EJL+03]. It offers a rich set of constructs to specify time requirements and constraints.

2.3.5 SysML/Marte

The UML profile for System Engineering (SysML) [OMG08] is an adopted OMG Specification to be used at the system level. SysML is based on a selected subset of UML constructs, called UML4SysML, and provides few new extensions amongst which *Refinement* and *Parametric* diagrams. The former helps making explicit system-level requirements and tracing their proposed implementations. The latter should be used to represent "non-causal" relationships amongst values of the system and possibly making explicit within the model, physical laws required for the design. "non-causal" is used here to denote equations in which variables are not assigned a particular role or direction. In $F = m \times \gamma$, no distinction is made between F, m and γ. A causal or functional interpretation would have considered m and γ as inputs and F as an output.

So, we can use this SysML construct to represent laws related to time, whether physical or logical. SysML recommends building a new "Constraint Block" for each new law and uses these blocks in so-called *parametric diagrams* to apply a law to relevant design values. In CCSL, there is a small number of identified relations among logical clocks. Consequently, we can easily construct a library of CCSL-specific constraint blocks. Figure 2.5 shows a CCSL specification expressed using SysML constraint

blocks and parametric diagrams.

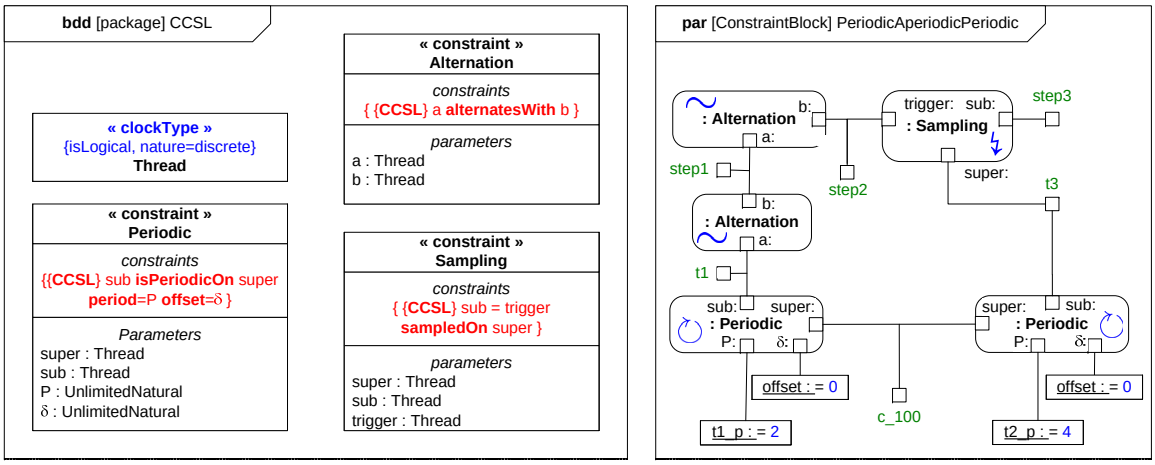

Figure 2.5: SysML parametric diagrams

The left-hand side part is an excerpt of the library. Three constraint blocks (Periodic, Alternation, Sampling) have been defined for each of the three CCSL relations introduced previously. Each constraint block has three compartments. The bottom one, called *parameters* contains typed *formal* parameters. The middle compartment, called *constraints*, contains the constraint itself that applies on the parameters. In our case, the constraint is defined in CCSL. However, this library is built once and for all, so end-users need not being entirely familiar with the concrete syntax and only need to be familiar with underlying concepts.

The right-hand side part models the following CCSL specification as a SysML parametric diagram:

```
t1 isPeriodicOn c_100 period=2 offset=0 // periodic thread t1
t3 isPeriodicOn c_100 period=4 offset=0 // periodic thread t3
t1 alternatesWith step1          // t1 executes step1
step1 alternatesWith step2       // step2 starts when step1 finishes
step3 = step2 sampledOn t3       // step3 samples the result from step2
```

In such a diagram, boxes are properties extracted from the model. Some of the properties are clocks ($t1$, $step1$ ...), some others have integer values ($offset$, $t1\_p$ ...). These properties may come from different diagrams and different blocks. The

rounded rectangles are usages of constraint blocks. Their ports, which represent parameters, are connected with properties using non-causal connectors. Being non-causal means that there is no input or output, whichever value is known causes the other to update. For instance, considering Alternation, if b is known, one can deduce (partially) a but if a is known, then one can deduce (partially) b. This example is further presented in Chapter 5.

2.4 The future of MARTE Time Model

MARTE v1.0 has been adopted in November 2009 and the first revision committee should release MARTE v1.1 by the end of 2010. The major modification in revision v1.1 concerning the time model is to authorize the stereotype «clock» to extend the metaclass «Event». The intent of this modification is to make it clear that any event of any kind (Call, Time, Send, ...) can be used as a logical time base in time specifications. They are no major modifications anticipated for release v1.2, but we shall continue our effort to disseminate MARTE and to increase the tool support. A second important requirements is to maintain and enhance the compatibility with SysML. The objective of next year being to provide SysML with a lighter time model that would be compatible with MARTE. SysML users do not seem ready to accept a textual language (like CCSL). Working with predefined domain-specific libraries as explained in Section 2.3.5 is one possible solution. An alternative would be to provide a better integration with the OMG's Object Constraint Language (OCL), more and more used, by offering a logically timed OCL. Timed OCL extensions already exist [F02] but propose to extend OCL with temporal logics constructs which is rather different from our objective.

Chapter 3

How does CCSL compare ?

This chapter compares CCSL to other closely related concurrent models: Petri Nets, Process networks, Signal. The comparison to Petri Nets and to Signal has been published at ISORC'09 [MA09] and the work on using the hierarchization mechanism has been published in the workshop MOBE-RTS [YTB+10]. The comparison to process networks have been accepted to be published in September 2010 in the proceedings of FDL 2010.

3.1 Petri Nets

3.1.1 Time Petri nets

MARTE Time model conceptually differs from Petri's work on concurrency theory [Pet87]. Petri's theory restricts coincidence to single points in space-time. In our model, the foundational relationship *coincidence* gathers *a priori* independent points (instants) to reflect design choices.

Petri nets have well-established mathematical foundations and offer rich analysis capabilities. They support true concurrency and can be used to specify some of our clock relations. However it is not possible to force two separate transitions to fire "at the same time", *i.e.,* to express coincidence. Thus, we use Time Petri net, the Merlin's extension [Mer74] that associates a time interval (two times a and b, with $0 \leq a \leq b$ and b possibly unbounded) with each transition. Times a and b, for transition t, are relative to the moment θ at which the transition was last enabled. t must not fire before time $\theta+a$ and must fire before or at time $\theta+b$ unless it is disabled (in case of conflicts) by the firing of another transition. Even with this extension, the specification of CCSL constraints is far from straightforward.

In our representation, each MARTE discrete-time pure clock $c = \langle \mathcal{I}_c, \prec_c \rangle$ is represented as a single transition c_t (called *clock transition*) of a Time Petri net. Instants of a clock are *firings* of the related transition. For a given initial marking and for a given firing sequence, there is an injective function $firing : CT \times \mathbb{N}^\star \to \mathbb{N}$, where CT is the set of clock transitions. $firing(c_t, i)$ is the time at which, the clock transition c_t fires for the i^{th} time in the firing sequence. We consider a Time Petri net as equivalent to a CCSL clock constraint, iff for all possible firing sequences and all clock transitions (other transitions do not matter), $firing$ preserves the ordering (Eq. 3.1).

$$\begin{aligned} &(\forall c1, c2 \in C)(\forall k1, k2 \in \mathbb{N}^\star) \\ &((c1[k1] \preccurlyeq c2[k2]) \\ \Leftrightarrow\ &(firing(c1_t, k1) \leq firing(c2_t, k2)) \end{aligned} \tag{3.1}$$

where $c1_t$ (resp. $c2_t$) is the clock transition associated with clock $c1$ (resp. $c2$). Note that, even though Time Petri nets can handle continuous time, we restrict our comparison to discrete-time clocks and therefore we consider the transition firing time as a natural number ($\in \mathbb{N}$).

3.1.2 Encoding CCSL operators in Time Petri nets

Precedence

CCSL strict precedence has a straightforward equivalent in Time Petri net (Fig. 3.1(a)). Note that the time interval $[1, \infty[$ of A prevents multiple firing of transition A at the same time. This condition can be weakened if needed. However, the time interval $[1, \infty[$ for B ensures the strictness of the relation: the i^{th} occurrence of B is strictly after the i^{th} occurrence of A in any valid behavior. The weak form of the precedence (depicted in Figure 3.1(b)) weaken the lower bound to allow simultaneous occurrences. In both cases, the place in-between the two transitions prevents B from ticking faster than A.

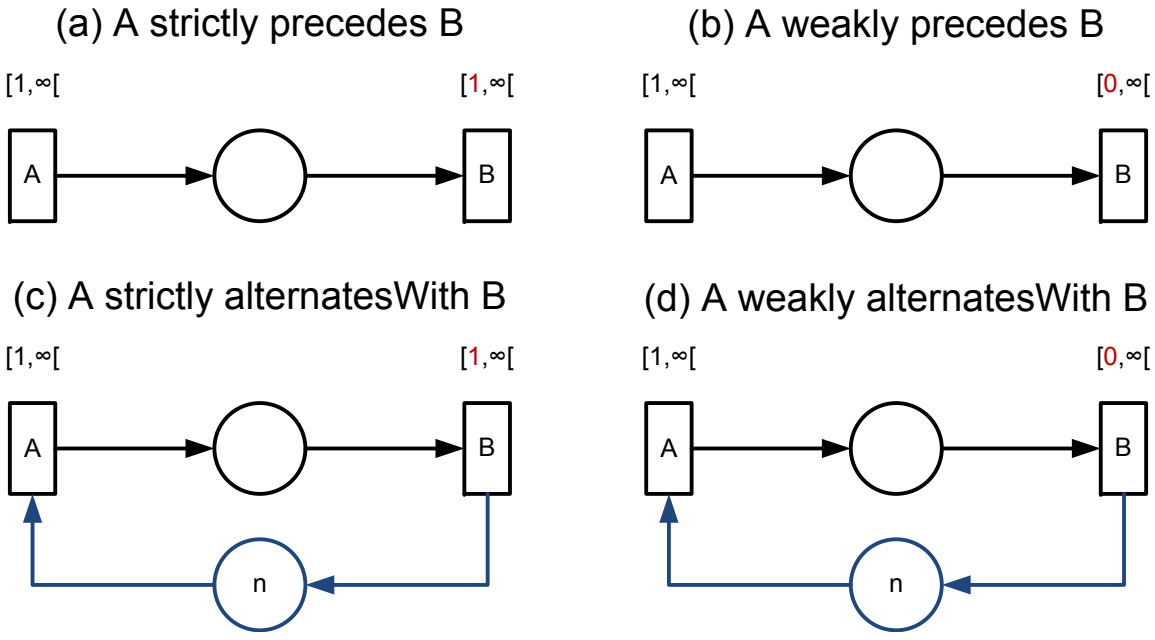

Figure 3.1: Precedence and alternation in Time Petri net

At the initial state (time=0) no transitions are enabled, only time can evolve. Then, transition A is enabled but there is no upper bound for its firing. Transition B will not become enabled before A fires. When A eventually fires, B becomes immediately enabled for the weak form and can fire "synchronously" with A. In the strict form, because of the time interval $[1, \infty[$, B must wait one instant before being enabled. Still, there is no upper bound.

Clearly both models are not bounded, only A can tick leading to (infinite) accumulation of tokens in the intermediate place. Alternation bounds the model by adding a cycle (see Fig. 3.1(c)-(d)). The number n of tokens in the newly introduced place gives the maximum advance A can have on B: default is 1.

Subclocking

Subclocking is achieved by using time intervals of $[0, 0]$ (see Fig. 3.2). This forces a transition (*e.g.,* B) to fire at the very same time it has been enabled. Conflicts may prevent the subclock (*e.g.,* B) from ticking but the subclock can never tick unless it has been enabled by the super clock (*e.g.,* A). As an example, we consider CCSL clock relation isPeriodicOn. Figure 3.2 models the following CCSL statement: A isSubclockOf B. Transition A must fire $\delta + 1$ times before anything can happen

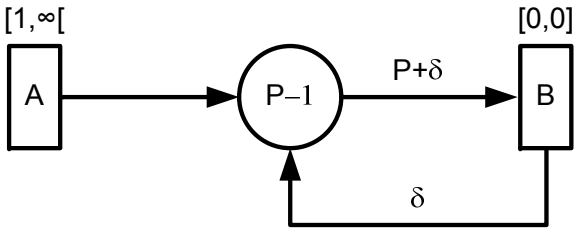

Figure 3.2: B isPeriodicOn A period=P offset=δ

to transition B. Then every P^{th} firing of transition A, B must fire synchronously because the time interval is $[0, 0]$. Such a solution is not compositional, since the time interval is relative to the time at which the transition is enabled, which depends on the marking of all incoming places.

Delay and Sampling

The binary delay of CCSL (*i.e.*, $B \boxed{=} A \$ \delta$) is equivalent to a periodic relation with $P = 1$ (see Fig. 3.2). The ternary delay (defer, $B = A \$_C \delta$), where the delay duration δ is relative to the ticks of a third clock (C), is more difficult to model.

Figure 3.3 attempts to represent $B \boxed{=} A \$_C 2$. Operator defer produces a clock B that is a subclock of C, which is represented with a time interval $[0, 0]$ for transition B. The same net (dashed part) can be unfolded as many times as needed depending on δ. However, this breaks the rule of having just one transition for each CCSL clock. And thus, such a model would require some external conditions to enforce that all transitions with the same label (here C) must necessarily fire at the same time instants. Other solutions described at ISORC'09 [MA09] rely on priorities but still require some kind of external scheduling policy that cannot be directly enforced with Time Petri nets. The major limitation being that it is not possible to enforce the simultaneous firing of different transitions. This comes from the initial design choice to restrict coincidence to single points in time.

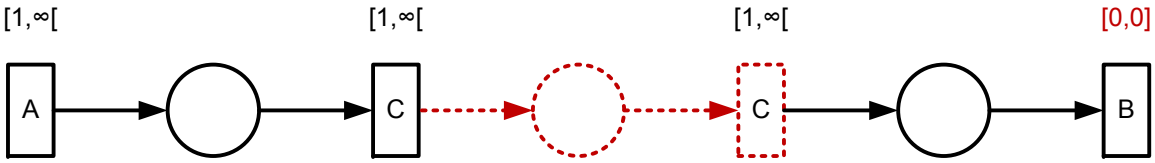

Figure 3.3: Operator defer: $B \boxed{=} A \$_C 2$

Note that sampling is natural in Petri nets. In that example, clock A is sampled on clock C to build a new clock B subclock of C but A is not necessarily in any subclocking relation with neither C nor B. Since CCSL operator *sampledOn* combines sampling and synchronizations, it requires similar assumptions than for the defer. See [MA09] for more details.

3.2 Process networks

3.2.1 Synchronous Data Flow

Process Networks [Kah74] is a common model for describing signal processing systems where infinite streams of data (unbounded FIFO channels) are incrementally transformed by processes executing in sequence or parallel. The global execution semantics of such systems is given by the set of local data dependencies between the processes (defined by the channel connections): *i.e.,* a process can only execute when his input channels contain enough data items. These local dependencies can be defined with CCSL by associating a *logical clock* with each process execution event and by translating each local data dependency into clock constraint rules. The rules would specify that, on a channel, the read of a data element (by the *slave* process[1]) must be preceded by the write of this data element (by the *master* process).

A common application of the process networks, in data-flow languages, uses a component-based approach for specifying the functionality of a system. "Actors" (or components) are the main entities. An actor consumes a fixed[2] amount of data ("tokens") from its input ports and produces a fixed amount of data on its output ports. A system is successively and hierarchically decomposed into a series of actors that are connected through data paths ("arcs"), which denote the *flow of data*.

Such basic assumptions favor static analysis techniques to compute a static schedule that optimizes a given criterion (*e.g.,* the buffer sizes) but limit the expressiveness of the specification. Additional features were introduced in many derivative languages to overcome these limitations. Several data-flow specification models have been proposed throughout the time. Most of these languages were designed around the *Synchronous Data Flow* (SDF) [LM87], proposed by Lee and Messerschmitt, or its multidimensional extension, *Multidimensional Synchronous Data Flow* (MDSDF) [ML02], designed to preserve the static properties for efficient implementations, while extending its expressiveness to cover a larger range of applications. SDF graphs are equivalent to Computation graphs [KM66], which have been proven to be a special

[1]The slave is on the arrow end of the arc

[2]Numerical values known at specification time

case of conflict-free Petri nets.

The multidimensional extension is essential for specifying complex data-flow applications where the data structures are commonly represented as multidimensional arrays or streams. On the other hand, this extension has an important impact on the actual execution order. Whereas the SDF model defines a strict ordering in time, MDSDF only defines a partial ordering: each dimension defines quasi-independent relations "past-future", as called in [ML02]. External constraints need to be introduced into the system to define a complete ordering in time. With MDSDF these additional constraints are hidden in the computation of a specific schedule optimized according to a specific criterion (*e.g.,* minimizing the buffer sizes, exploiting maximum of parallelism).

Array-OL [GDB09] takes the concept of multidimensional order even further, by completely mixing space and time into the data-structures at the specification level: a single assignment of multidimensional arrays with possibly infinite dimensions (maximum one by array) defines the set of data values that will transit through the system. Data dependencies between uniform and repetitive patterns of data are defined. The global order depends on the sets of depending pairs of actor executions, where two actor instances are execution-dependent if the patterns produced/consumed share at least a common data element. In such a MoCC, a total order between executions cannot be deduced unless additional environment constraints are specified.

3.2.2 A CCSL library for SDF

In [MDAd10], data-dependencies defined by SDF arcs are expressed as CCSL relations. The actor executions are modeled by logical clocks. Each clock instant denotes one execution of the related actor. Logical clocks are also used to model read/write operations on the arcs. The CCSL rule associated with an arc represents a conjunction of three relations, as follows:

1. A packet-based precedence between the logical clock *read*, representing reading instants from the channel, and the logical clock *slave*, representing the execution of the slave. Eq. 3.2 states that w_{rd} read events are needed before an actor can

execute. The strictly positive integer w_{rd} represents the slave input weight.

2. Each actor execution is followed by w_{wr} write events on each of the output arcs (see Eq. 3.3). w_{wr} represents the master output weight.

3. For a given arc, the i^{th} tick of *write* must precede the i^{th} tick of *read*. When *delay* tokens are initially available in the queue, the i^{th} read operation uses the data written at the $(i - delay)^{th}$ write operation, for $i > delay$ (see Eq. 3.4).

$$\begin{aligned}
&\mathsf{def}\ arc(\mathsf{clock}\ master,\ \mathsf{clock}\ write,\ \mathsf{int}\ w_{wr},\ \mathsf{int}\ delay,\\
&\qquad \mathsf{clock}\ slave,\ \mathsf{clock}\ read,\ \mathsf{int}\ w_{rd}) \triangleq \\
&\qquad (read\ \mathsf{by}\ w_{rd})\ \boxed{\prec}\ slave && (3.2)\\
&\qquad master\ \boxed{=}\ (write\ \mathsf{by}\ w_{wr}) && (3.3)\\
&\qquad write\ \boxed{\prec}\ (read\ \$\ delay) && (3.4)
\end{aligned}$$

3.2.3 Discussion and perspectives

The data dependencies between two actors at the ends of an arc are expressed in this proposition by CCSL clock constraints between element-wise production/consumption on this arc. For SDF models with actors that produce and consume a larger number of tokens by execution, this approach explodes in size at simulation. Moreover, the essential aspects between the relative executions of actors would be completely negligible compared to the overwhelming information concerning token writings and readings. Therefore, we have proposed[3] a new way to translate data dependencies induced by a SDF arc into CCSL relations between actor executions, without going through the element-wise write/read operations. The read tokens/execute actor/write tokens operations are abstracted by a single atomic event. Expressing the execution dependency between a master and a slave actor (linked by an arc) means identifying the minimum number of executions of the master actor needed to enable the slave

[3]In a paper actually under a review process

actor execution. This requires a local scheduling the result of which can be expressed with CCSL operator filteredBy. The details are not given here.

In this context, CCSL is used as a language based on logical time to define a timed causality semantics for models. Syntactic models are complemented with a semantic model described in CCSL. The behavior of a system is thus expressed as a formal specification encoding the set of schedules corresponding to a correct execution. SDF is particularly well-adapted to build a dedicated CCSL library since the semantics of a system is fully defined by the set of local execution rules imposed by the data dependencies. Part II shows other examples that have been addressed with CCSL.

In [MDAd10], fine grain element-wise data dependencies of SDF were encoded in CCSL. During his postdoctoral year, Calin Glitia has proposed to express directly the actor activations by processing local data dependencies and its algorithm has been implemented and integrated as a new plug-in in TimeSquare. For more complex languages, translating the data dependencies into execution dependencies that can be expressed by the CCSL language implies more complex computations. Extension to MDSDF is straightforward. However, it is more complex to encode the execution rules of ARRAY-OL or other polyhedral models where data dependencies are defined relatively to regular but arbitrary shaped sub-arrays.

Beyond that simple encoding in CCSL, we think that CCSL can complement data-flow languages by reflecting the actual (partial) ordering in which data are processed, enter or exit the system. Indeed, whereas data flow languages focus on the different production and consumption rates of data, CCSL focus on logical ordering of actions. Typically, such multidimensional ordering is just partially defined in MDSDF where dimensions are *a priori* independent. However, when computing a particular static scheduling to optimize some criteria (*e.g.,* buffer sizes), decisions are taken on the actual ordering of data. Different algorithms take different ordering choices. Our point is that this choice should be part of the model and can be made explicit with CCSL instead of being hidden in the chosen scheduling algorithm. In other words, there is an opportunity here to define scheduling algorithms that take benefit from a joint use of the time and the repetitive structure models of MARTE.

3.3 Polychronous languages

3.3.1 Signal

SIGNAL is a synchronous dataflow language, which is based on synchronized dataflows (flows + synchronization). Variables (*e.g.,* x) are called *signals* and represent an infinite typed sequence, which is mapped onto the *logical time* indexed by natural numbers, *i.e.,* x is actually $(x_\tau)_{\tau\in\mathbb{N}}$. The symbol $\perp$, which represents the absence of the signal at certain instant on the logical time, expands the domain of signal. A signal has an associated clock (not to be mistaken with CCSL clocks) indicating the set of instants when the signal is present. A *process* is considered as a program that is composed of a system of equations over signals and an interface. As in MARTE, the physical amount of time between two values is not relevant.

SIGNAL allows the specification of *multiclock/polychronous* systems, in which a process can be deactivated while other processes are still activated. Two kinds of operators are defined in SIGNAL: monochronous and polychronous. The former operates on signals with the same clock, *i.e.,* signals that are always present at the same time. The latter handles signals with different clocks. In addition, the SIGNAL formal model allows partial and nondeterministic specifications. The model also supports a design methodology which goes from specification to implementation, from synchrony to asynchrony. We only consider here the time structure of MARTE and relations on instants, we do not use the labeling functions. So CCSL clocks are very similar to signals and clocks compare to SIGNAL clocks (or *pure signals*, type event). CCSL clock relations compare to SIGNAL polychronous operators. In this document, we never discuss equivalent for SIGNAL monochronous operators that would work on labels associated with instants rather than on the time structure itself.

3.3.2 Encoding CCSL operators in Signal

Precedence

Encoding the unbounded CCSL Precedence requires to use a local integer (line 6) that counts occurrences of signals A and B (lines 3 and 4) and allows B to tick only when the counter is greater than zero (line 6), but not necessarily always.

```
1 process strictlyPrecedes =
2  ( ? event A,B )
3  (| zcounter := counter$1 init 0;
4   | counter := zcounter + 1 when A when not B
5   |    default zcounter - 1 when B when not A default zcounter
6   | B ^> when counter>0
7   |) where integer counter init 0, zcounter end;
```

Even though the alternation can be seen as the composition of two precedence relations, it is more efficient to encode a finite automaton (see Fig. 3.4).

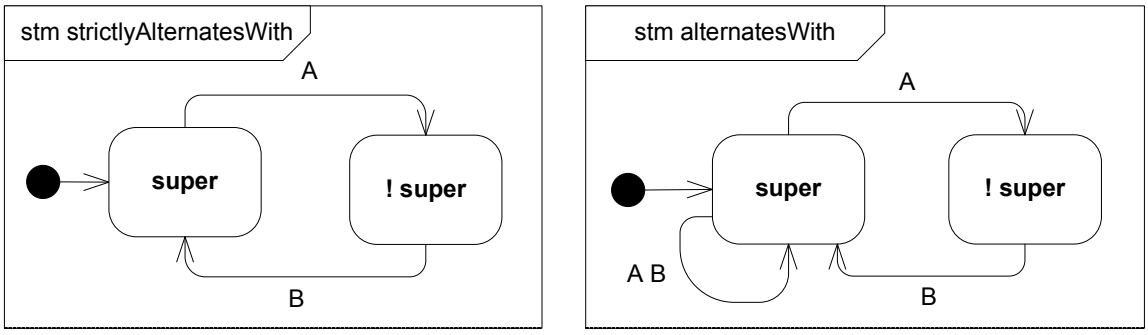

Figure 3.4: CCSL alternatesWith encoded as an automaton

When two clocks strictly alternate, there is a super clock, more frequent than both A and B (the relation is *endochronous*). To implement such a relation in SIGNAL, one just need to build the common super clock explicitly. In the following SIGNAL implementation, line 1 declares a concurrent process and line 2 declares its two pure input signals. Line 3 builds a two-state automaton (see Figure 3.4, left part) that

alternates between the two states. The Boolean signal super is local (see line 6) and alternatively takes the value *true* and $false$, starting with *true*. Signal A is present when and only when super is *true* (line 4). Signal B is present when and only when super is $false$ (line 5).

```
1 process strictlyAlternatesWith =
2  ( ? event A,B )
3  (| super := not (super $ 1 init false)
4   | A ^= when super
5   | B ^= when not super
6   |) where Boolean super end;
```

The weak form is more complex because either A and B simultaneously occurs, or A occurs alone and B should occur alone in the future. Note that B cannot occur alone when super is *true*. The implementation below directly implements the state machine shown on the right-hand side of Figure 3.4. There are still two states encoded with the local Boolean signal super. The state can also change when either A or B occurs. The signal union is denoted by the operator ^+ in Signal (line 3). When B occurs, then the next state (nextsuper) is necessarily *true* (line 5), whatever the current state and whether or not A occurs. When B does not occur the next state is $false$ (line 4). Conversely, A must and can only occur when super is true (line 7). When super is $false$, B must occur but B can also occur when *super* is true. Line 8 reads that B is more frequent than when super is $false$. The only other possible case is when super is *true* because of the signal union in line 3.

```
process alternatesWith =
 ( ? event A,B )
 (| nextsuper ^= super ^= A ^+ B
  | nextsuper := false when not B
                        default true
  | super := nextsuper $ 1 init true
  | A ^= when super
  | B ^> when not super
  |) where Boolean super,nextsuper end;
```

Subclocking

Expressing subclocking is straightforward in SIGNAL (operator^<) and CCSL operator filteredBy is very close to the SIGNAL operator when, except that when uses a Boolean condition whereas filteredBy uses a binary word. Encoding CCSL operator isPeriodicOn is one simple application.

```
process isPeriodicOn =
  { integer offset,period }
  ( ? event A, B )
  (| nb ^= B
   | zi := nb $ 1
   | nb := ((zi + 1) when zi /= (period-1))
                      default 0
   | ^A ^= when zi=0
   |) where
       integer zi init -offset, nb
     end;
```

Not surprisingly, all coincident-based operators have almost direct equivalent in synchronous languages in general and in SIGNAL. It is not always the case for precedence-based operators for which it may be tedious. Even when the languages can be twisted to model such constraints, the compiler is not always able to find a solution. Consequently, it is really useful to have a specification language that simply supports various concepts even if several implementation languages must be combined to find possible solutions.

Delay and Sampling

SIGNAL delay operator ($) is monochronous and CCSL binary operator $ is equivalent. However, CCSL operator defer is a polychronous operator that has no direct equivalent in SIGNAL. defer samples a clock *inp* on the n^{th} occurrence of another clock *clk*. CCSL operator sampledOn is a defer with a duration of 1. We describe the encoding of this operator in SIGNAL. Its *weak form* is more difficult to implement since it implies instantaneous reactions. The following SIGNAL process counts the number of occurrences of *inp* between two successive occurrences of *clk*. A sampling occurs where there is at least one occurrence of *inp* (zc not equal to 0, zc/=0).

```
1 process strictlySampledOn =
2   (? event inp, clk ! event outp )
3   (| c ^= zc ^= inp ^+ clk
4    | zc := c $ 1 init 0
5    | c := 1 when clk when inp
6              default 0 when clk
7              default zc+1 when inp
8    | outp := when zc/=0 when clk
9    |) where integer c, zc end;
```

The weak form is similar but if the input event (*inp*) occurs simultaneously with the sampling clock (*clk*), *i.e.,* it is not strictly future, then it must be sampled. This requires to be one more step ahead (*zzc*) (see [MA09] for details).

3.3.3 Hierarchization of CCSL clock constraints

As a collaborative work with INRIA team-project ESPRESSO, Huafeng Yu has studied ways to use the SIGNAL tool suite and its clock calculus engine to analyze CCSL specifications [YTB+10]. CCSL aims at providing a general time model with regards to clock relations. However it is not yet supported by many tools. On the contrary, SIGNAL comes with plenty of analysis tools for clock relations (Polychrony). Hence it is a promising approach to benefit from the various tools of Polychrony to enhance the analysis capability of TimeSquare. Nevertheless, the expressiveness of the two languages, *i.e.,* SIGNAL and CCSL, is different as they are not faced with the same problems. The main differences between TimeSquare/CCSL and Polychrony/SIGNAL are summarized here:

- CCSL aims at providing a more generic time model than SIGNAL. Both dense time and discrete time are supported in CCSL, whereas only logical time is allowed in SIGNAL. Hence, it is necessary to map dense and discrete time of CCSL onto the logical time of SIGNAL. For instance, the dense time is mapped onto the discrete time through sampling or discrete observation. Then, the discrete time is mapped onto the logical time in a natural way.

- CCSL allows the specification of clock relations with numerical properties. Some of them can be also specified in SIGNAL. However, some numerical properties, *e.g.,* duration, are not well supported by the code generation of SIGNAL programs. On the contrary, SIGNAL arithmetic operations on numbers are not supported in CCSL.

- Asynchronous clock constraints are more easily specified and addressed in CCSL than in SIGNAL. TimeSquare provides a constraint solver that addresses these constraints in a nondeterministic way, thus it allows nondeterminism in the simulation. SIGNAL also allows the specification of asynchronous clock constraints, whereas the SIGNAL compiler refuses direct code generation for these nondeterministic constraints. Hence, a valid specification of TimeSquare is not always accepted by the SIGNAL compiler for code generation.

The main expected benefit of integrating Polychronous and TimeSquare is to be able to use the clock *hierarchization* of Polychrony to detect determinism in CCSL specifications and then use SIGNAL facilities for code generation or deterministic simulation. Non deterministic specifications can still be analyzed with TimeSquare classical mechanisms.

One of the main objectives of the hierarchization is to determine an endochronous clock system E by analyzing a SIGNAL program S. An endochronous system implies a unique root node in the hierarchization tree. This endochronous system ensures the deterministic scheduling of arriving events only according to the internal signal state and structure of S. In Polychrony, E can be used to generate code.

However, it is not always easy to build an endochronous system from S since S can be polychronous. Clocks can be completely independent, *i.e.,* no communication occurs between processes. Clocks can also have constraints between them, but no synchronous relationships. For instance, when using CCSL relation *sampledOn*. In this case, it induces nondeterminism. In TimeSquare, the user can choose amongst a set of possible simulation policies (*e.g.,* random, as soon as possible, priority-based) to select one solution out of the many possible ones. In Polychrony, code cannot be generated for nondeterministic specifications.

There are several solutions to obtain a deterministic behavior. The first solution offered by Polychrony consists in adding supplementary clocks to endochronize polychronous clocks. For endochronous systems, the code generation is possible. It is therefore complementary to the TimeSquare constraint solver. Unfortunately, it is not always possible to find appropriate supplementary clocks for polychronous systems. Moreover, once these supplementary clocks are integrated into the system, the compositionality of these systems cannot be ensured.

3.4 Perspectives

We have compared CCSL to other concurrent models that are often used in the field. The comparison work is not over and as the scope of CCSL increases, the comparison targets augment. Petri nets and process networks are natively untimed and time

comes as a decoration to describe mostly non functional properties. In CCSL time, causal and chronological relations are natively built to emphasize on the functional role that time can play in a specification. Signal, even though very close to CCSL, appeared as of complementary use and there is an ongoing effort to combine them.

Part III describes a possible use of CCSL specifications as a base to verify VHDL implementations. As such CCSL appears to have overlapping objectives with temporal logics in general and the Property Specification Language (PSL) [PSL05] in particular. A comparison with temporal logics in on-going and has to be completed. Preliminary results showed that some CCSL operators (like precedence) could not be encoded with temporal logics, whereas some (Linear Temporal Logics) LTL operators (*e.g.,* UNTIL) could not be encoded in CCSL.

In any cases, our goal is to integrate CCSL in a design flow and to show how it can complement other formal languages and models.

Part II

Modeling

This part gives usage examples of the MARTE time model selected from projects in which we were involved and addressing several application domains.

The work described in chapter 4 was conducted in the context of the ANR RNTL MeMVaTEx (Méthode de Modélisation pour la Validation et la Traçabilité des Exigences) project (2006-2009), led by Siemens VDO. The main goal of MeMVaTEx was to define a UML-based methodology for the design of embedded software for the automotive domain. It focused on the validation and traceability of requirements. The work has been presented at ISORC'09 [MPFA09] and an extended version is available as a research report [MPA08].

Chapter 5 describes a work that was started during the elaboration of MARTE in order to use MARTE as a UML profile for AADL[4] (Architecture & Analysis Description Language) [FGH06]. AADL is an architecture description language (ADL) [MT00] adopted by the SAE (Society of Automotive Engineering) that offers specific support for schedulability analysis. It also considers classical computation (periodic, sporadic, aperiodic) and communication (immediate/delayed, event-based or timed-triggered) patterns. However, it departs from EAST-ADL because it explicitly considers the execution platform to which the application is allocated. Our illustration uses MARTE (and notably its allocation subprofile) to build a model amenable to architecture exploration and schedulability analysis. The expression in CCSL of immediate and delayed communications in case of periodic tasks has been presented at FDL'07 [MAdS07]. The merging of event-based and time-triggered aspects has been presented at FDL'08 [MdSR08]. The effort on the convergence between MARTE and AADL is still on-going and is partly conducted in the context of a 3-year FUI project, called Lambda (Libraries for Applying Model-Based Development Approaches), which started in 2008.

[4] `http://www.aadl.info`

Chapter 4

The automotive domain

We first consider an example from the automotive domain. We build a CCSL library to express formal time requirements. The operational semantics of CCSL is exploited to make the requirements executable.

4.1 An ADL for automotive software: East-ADL2

There is a stringent need to master the growing complexity of automotive electronic architectures (and thus software), which has taken a tremendous part in the automobile design flow. Many initiatives have proposed to tackle this growing complexity by providing a common architecture between suppliers and manufacturers. These initiatives have involved the entire automotive electronic value system, ranging from semiconductor industries, tool and software vendors through tier-one suppliers to the car makers themselves. Such a broad collaboration requires a common methodology and the definition of standards and interchange formats to support tool interoperability.

Since 2003, the main effort in that path from the industry has lead to AutoSAR (AUTomotive Open System ARchitecture)[1], an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and

[1] `http://www.autosar.org`

tool developers. There have also been several academic/industry European projects following the same line and that have fostered some of the solutions actually adopted by AutoSAR. One of this project is East-EEA [The04], an ITEA European project (2001-2003), whose main visible result was to propose a dedicated UML profile called East-ADL (Electronic Architecture and Software Tools, Architecture Description Language). To integrate proposals from the emerging standard AUTOSAR and from other requirement formalisms like SysML [Wei08, OMG08], a new release called East-ADL2 [CCG+07, The08] has been proposed by the ATESST project (Advancing Traffic Efficiency and Safety through Software Technology)[2]. In this section, we abusively refer to both versions under the name East-ADL.

Structural modeling in East-ADL covers both analysis and design levels. In this paper the focus is on the analysis level and especially on timing requirements. We build a CCSL library for expressing the semantics of East-ADL timing requirements. Their semantics is left informal in East-ADL specification ([The08], chapter 14) and we had to disambiguate some of their definitions to build our CCSL model. By building this library we make East-ADL requirement specifications executable and allow the use of TimeSquare to execute and animate UML models annotated with East-ADL stereotypes. The formal semantics can then lead the transformation to dedicated analysis tools (such as SymTA/S [HHJ+05, PEP02], MAST [HGGM01], the Real-Time Calculus toolbox [TCN00]), whereas a purely syntactic model transformations would prevent an actual interpretation of analysis results on the UML model.

4.1.1 Timing Requirements

East-ADL requirements extend SysML requirements and express conditions that must be met by the system. They usually enrich the functional architecture with extra-functional characteristics such as variability and temporal behavior. We focus here on the three kinds of timing requirements (Fig. 4.1):

[2] `http://www.atesst.org/`. ATESST and its follow-up ATESST 2 are STREP projects funded by the European Commission.

1. DelayRequirement that constrains the delay "from" a set of entities[3] "until" another set of entities. It specifies the temporal distance between the earliest event occurrence on the "from" entity and the latest event occurrence on the "until" entity. It is used to specify end-to-end delays;

2. RepetitionRate that defines the inter-arrival time of data on a port or the triggering period of an elementary ADLFunction;

3. Input/OutputSynchronization that expresses a timing requirement on the input/output synchronization among the set of ports of an ADLFunction. It should be used to express the maximum temporal skew allowed between input or output events or data of an ADLFunction.

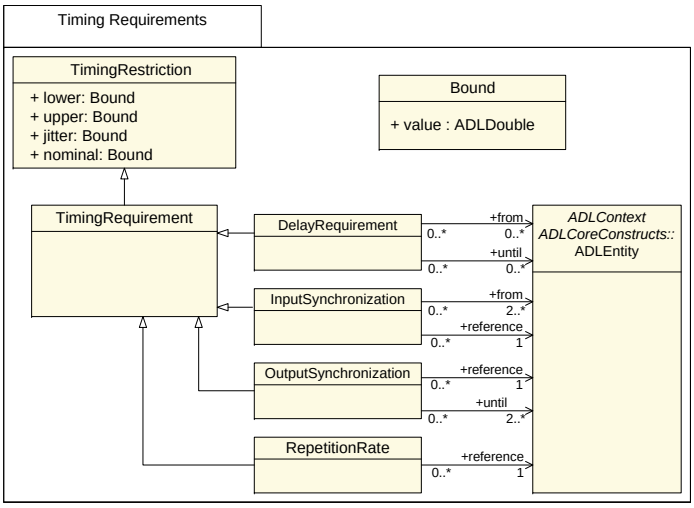

Figure 4.1: The metamodel of East-ADL Timing Requirements.

Timing requirements specialize the meta-class TimingRestriction, which defines bounds on system timing attributes. The timing restriction can be specified as a *nominal* value, with or without a *jitter*, and can have *lower* and *upper* bounds. The jitter is the maximal positive or negative variation from the nominal value. A bound is a real number associated with an implicit time unit (ms, s...).

[3]Entity is the official terminology of East-ADL. In practice, it mainly refers in that case to a port of a software component.

4.1.2 Example: An ABS controller

As an illustration, we consider an Anti-lock Braking System (ABS). This example and the associated timing requirements are taken from the ATESST report on East-ADL timing model [JLF08]. The ABS architecture consists of four sensors, four actuators and an indicator of the vehicle speed. The sensors (i_{fl}, i_{fr}, i_{rl}, i_{rr}) measure the rotation speed of the vehicle wheels. The actuators (o_{fl}, o_{fr}, o_{rl}, o_{rr}) indicate the brake pressure to be applied on the wheels. The FunctionalArchitecture is composed of FunctionalDevices for sensors and actuators and an ADLFunctionType for the functional part of the ABS. An ADLOutFlowPort provides the vehicle speed (speed).

The execution of the ABS is triggered by the successive occurrences of event R (Fig. 4.2), a RepetitionRate. Parameter Ls represents the latency of sensor sampling. At each cycle, the values acquired by the four sensors must arrive on the respective input ADLFlowPorts within the delay Jii (InputSynchronization). A similar OuputSynchronization delay Joo is represented on the output interface side. Lio represents the delay from the earliest event occurrence amongst the four input ports of the ABS until the latest event occurrence amongst the four output ports, it is a DelayRequirement in East-ADL terminology. The sampling interval of the sensor is given by parameter H. All these parameters are modeled by timing requirements characterized by timing values or time intervals with jitters.

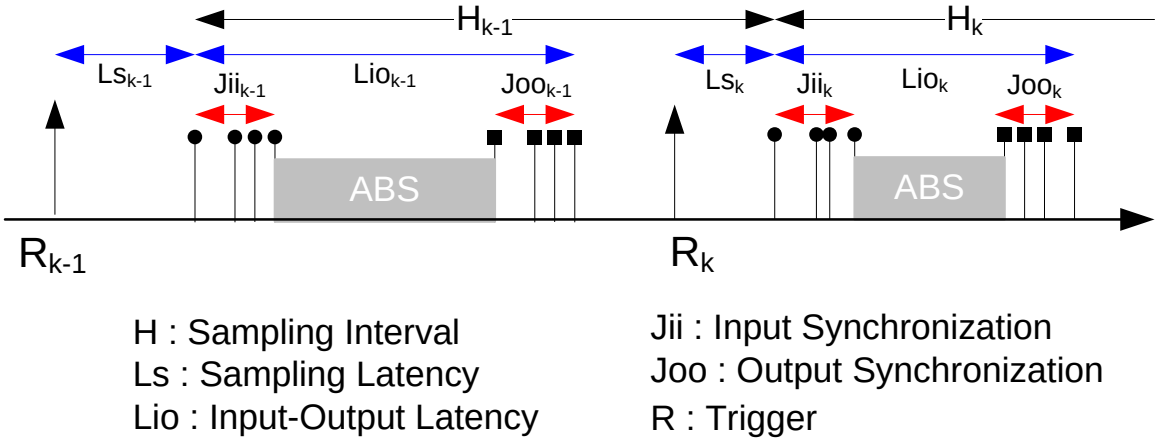

Figure 4.2: Timing requirements for the ABS

4.2 A CCSL library for East-ADL

East-ADL introduces a vocabulary specific to the sub-domain considered (delay requirement, input/output synchronization, repetition rate). These time requirements can be modeled simply by composing CCSL relations. For each one of the three timing requirement kinds, we build a CCSL relation definition, stored in a library.

4.2.1 Applying the UML profile for Marte

The ABS function is modeled in UML (Fig. 4.3) and some model elements (TimedElements) are selected to apply the CCSL clock constraints. The reaction of a timed element is dictated by the clock associated with it. For instance, sensor i_{fl} is a timed element associated with clock i_{fl}. Ticks of clock i_{fl} should be interpreted as a data acquisition from the sensor. Similarly when clock o_{fl} ticks, actuator o_{fl} emits data.

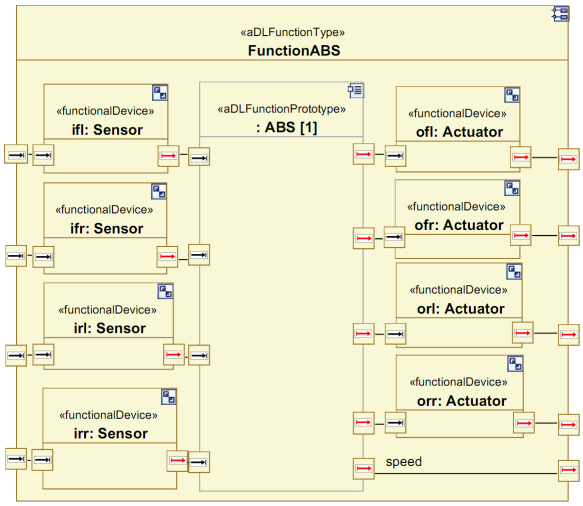

Figure 4.3: Example of the ABS

In the following, we explain how the three different kinds of timing requirements defined in East-ADL can be modeled with CCSL constraints.

4.2.2 Repetition rate

A RepetitionRate concerns successive occurrences of the same event (data arriving to or departing from a port, triggering of a function). In all cases, it consists in giving a nominal duration between two successive occurrences/instants of the same event/clock. We build a CCSL relation definition called *repetitionRate* with three parameters: *element*, *rate* and *jitter*. *element* is the clock on which a repetition rate is applied. *rate* is an integer, the actual repetition rate. *jitter* is a real number, the jitter with which the repetition rate is expressed.

$$\begin{aligned}
&\mathsf{def}\ repetitionRate(\mathsf{clock}\ element,\ \mathsf{int}\ rate,\ \mathsf{real}\ jitter) \triangleq \\
&\quad \mathsf{clock}\ c_1 \boxed{=} idealClk\ \mathsf{discretizedBy}\ 0.001 && (4.1)\\
&\quad |\ element\ \mathsf{isPeriodicOn}\ c_1\ \mathsf{period}\ rate && (4.2)\\
&\quad |\ element\ \mathsf{hasStability}\ jitter/rate && (4.3)
\end{aligned}$$

This relation definition involves three CCSL constraints. For the duration to be specified in seconds (time unit s), we use the clock *idealClk* defined in the MARTE time library (Section 2.1). The CCSL expression discretizedBy discretizes idealClk and defines a chronometric discrete clock c_1 so that the distance between two successive instants of c_1 is 0.001 s (Eq. 4.1). The unit (here s) is the default unit defined for idealClk and therefore c_1 is a 1 kHz chronometric clock. Eq. 4.2 uses the CCSL expression isPeriodicOn to undersample c_1 and build another clock *element*, *rate* times slower than c_1. Eq. 4.2 is equivalent to $element \boxed{=} c_1 \blacktriangledown (1.0^{rate-1})^{\omega}$.

Finally, Eq. 4.3 expresses the jitter of the repetition rate. The CCSL constraint hasStability states that the clock *element* is not strictly periodic: a maximal relative variations of $jitter/rate$ is possible on its period.

Now, back to the ABS example. One time requirement of the ATESST example specifies that the ABS function must be executed every 5 ms with a maximum jitter of 1 ms. If *abs.start* is the clock that triggers the execution of the function *ABS*, then $repetitionRate(f.start, 5, 1)$ enforces this requirement. A jitter of 1 ms for a nominal period of 5 ms corresponds to a stability of 20 %.

4.2.3 Delay requirements

A DelayRequirement constrains the delay between a set of inputs and a set of outputs. At each iteration, all inputs and outputs must occur. So, defining a delay requirement between two model elements means constraining the temporal distance between the i^{th} occurrences of their respective events. In the ATESST example, a delay requirement is used, for instance to constrain the end-to-end latency of ABS: at each iteration, the distance between the reception of the earliest input and the emission of the latest output must be less than 3 ms. Consequently, we define a CCSL clock relation named *distance* with three parameters: *begin*, *end* and *duration*, so that the distance between the i^{th} occurrence of *begin* and the i^{th} occurrence of *end* must be less than *duration* ms. When a better precision than the ms is required, a 10 kHz chronometric clock can replace the 1kHz one (Eq. 4.4). *delayedFor* (Eq. 4.5) expresses a pure delay where the delay duration is counted in number of ticks of c_{10}.

$$\begin{aligned} &\mathsf{def}\ \mathit{distance}(\mathsf{clock}\ \mathit{begin},\ \mathsf{clock}\ \mathit{begin},\ \mathsf{int}\ \mathit{duration}) \triangleq \\ &\quad \mathsf{clock}\ c_{10}\ \boxed{=}\ \mathit{idealClk}\ \mathsf{discretizedBy}\ 0.0001 && (4.4) \\ &\quad |\ \mathit{end}\ \boxed{\prec}\ (\mathit{begin}\ \mathsf{delayedFor}\ \mathit{duration}\ \mathsf{on}\ c_{10}) && (4.5) \end{aligned}$$

As we need to model the arrival of the earliest input and of the latest output, we use the Kernel CCSL expressions inf and sup. $inf(a, b)$ is the greatest lower bound of a and b for the precedence relation $\boxed{\preccurlyeq}$ and $sup(a, b)$ is the lowest upper bound.

$$\begin{aligned} &\mathsf{clock}\ i_{inf}\ \boxed{=}\ \mathsf{inf}(i_{fl}, i_{fr}, i_{rl}, i_{rr}); && \mathsf{clock}\ i_{sup}\ \boxed{=}\ \mathsf{sup}(i_{fl}, i_{fr}, i_{rl}, i_{rr}); \\ &\mathsf{clock}\ o_{inf}\ \boxed{=}\ \mathsf{inf}(o_{fl}, o_{fr}, o_{rl}, o_{rr}); && \mathsf{clock}\ o_{sup}\ \boxed{=}\ \mathsf{sup}(o_{fl}, o_{fr}, o_{rl}, o_{rr}); \end{aligned}$$

With these four new clocks, the specification of the end-to-end latency becomes $distance(i_{inf}, o_{sup}, 30)$. Similarly, input (resp. output) synchronizations are specializations of a delay requirement. An input synchronization delay requirement for the function ABS bounds the temporal distance between the earliest input and the latest

input (specified by *Jii* on Figure 4.2). $distance(i_{inf}, i_{sup}, 5)$ enforces an input synchronization of 0.5 ms. Likewise, $distance(o_{inf}, o_{sup}, 5)$ enforces an output synchronization of 0.5 ms (see *Joo* on Fig. 4.2).

4.3 Analysis of East-ADL specification

TimeSquare proposes menus dedicated to East-ADL requirements, allowing an interactive specification of East-ADL models. The menus build an internal model of the specification as well as a UML MARTE model. The internal model is then transformed into either a pure East-ADL model or a pure CCSL specification. East-ADL models can be used by East-ADL-compliant tools. CCSL specifications are analyzed by the TimeSquare clock calculus engine to detect inconsistent specifications or to execute the UML model. The execution trace can be dumped as a VCD file or can drive the animation of the UML model. Figure 4.4 shows a trace example resulting from a complete specification of the ABS. This execution exhibits a violation of the specification showing that all the computations involved (ABS, sensors and actuators) cannot be executed within the imposed 5 ms repetition rate.

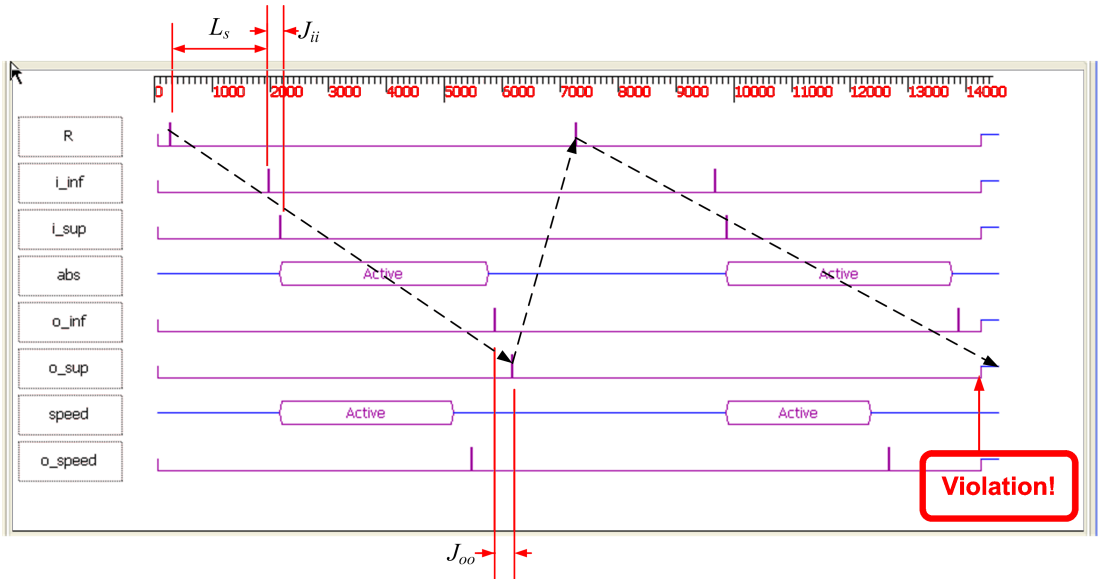

Figure 4.4: Executing the East-ADL specification of the ABS with TimeSquare

4.4 Perspectives

In November 2009, the official timing model of AUTOSAR has been released [AUT09]. A similar work should be conducted on this specification. It seems that CCSL is well-adapted for this purpose. AUTOSAR timing model is based on the observation of events and their occurrences. Timing requirements are specified by applying timing constraints on event occurrences. AUTOSAR proposes a classification of events, this classification exactly identifies the kinds of events that can be considered. For instance, when working at the level of a software component, AUTOSAR considers three kinds: runnableEntityActivated, runnableEntityStarted, runnableEntityTerminated. In CCSL, the nature of events is not given, so nothing prevents us from assigning one CCSL clock to an event for each kind of events, for each runnable entity. In MARTE, the UML defines what can be considered as an event and new events are introduced by some stereotypes of the MARTE time model. For instance, «TimedProcessing» introduces a start and a finish event for any action, behavior or message. Nothing is actually proposed to distinguish the activation from the start. Consequently, providing a full mapping from MARTE to AUTOSAR requires a careful analysis of both specifications.

Looking at AUTOSAR Requirements on Timing Extensions ([AUT09], section 1.6), several concepts from East-ADL are directly reused. For instance, the requirement RSTM002 states that "*The AUTOSAR templates shall provide the means to describe timing constraints, such as software and hardware latency, input/output delay, synchronization and runnable execution order constraints with clearly defined semantics.*". There are also some novelties that are perfectly within the scope of CCSL, like the one considered by RSTM008: "*The AUTOSAR templates shall provide the means to describe multiple asynchronous clocks/time bases and their interrelation.*".

Even though in AUTOSAR and East-ADL, all the timing requirements are expressed relatively to the physical time, the analysis tools generally ignore the units. The important information is the relative rates between repetitive event occurrences, and therefore the library could be rewritten by ignoring the figures related to physical time and simply relying on pure logical clocks.

Chapter 5

The avionic domain

In this second example, we consider AADL and use a combination of MARTE and CCSL to build its software components, execution platform components, express the binding relationships. Our intent is to allow a UML MARTE representation of AADL models so as UML models can benefit from the analysis tools (mainly for schedulability analysis) that accept AADL models as inputs.

The UML profile for AADL is defined as a subset of MARTE. It is included in the OMG MARTE specification ([OMG09a], Annex A.2). Part of the discussion presented here has now been included in the MARTE annex. However, the focus here is not so much on the relation with MARTE but rather with CCSL. Indeed, CCSL can explicitly model AADL execution patterns (periodic, sporadic, aperiodic) and communication schemes (delayed and immediate). These computation patterns and communications schemes are not compositional (*i.e.,* the semantics of a compound cannot be directly inferred from the semantics of the components) and a careful analysis [FH07] of a compound is required to understand the emerging semantics. CCSL can be used to model explicitly this emerging semantics. To illustrate the discussion, we use an example (see Fig. 5.1) taken from a report introducing latency analysis with AADL [FH07].

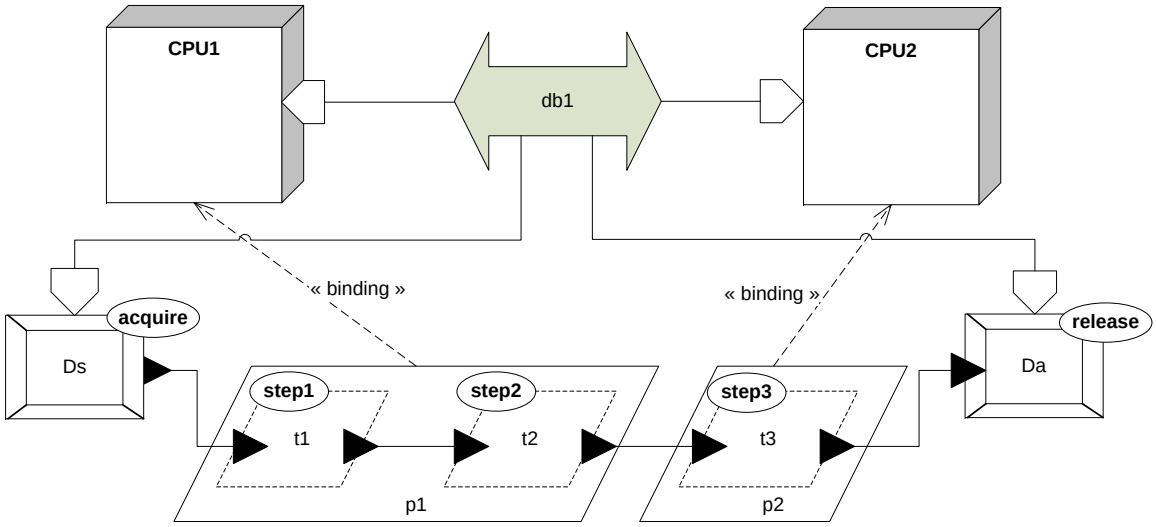

Figure 5.1: The example in AADL

5.1 Architecture & Analysis Description Language

5.1.1 Modeling elements

AADL supports the modeling of application software components (thread, subprogram, and process), execution platform components (bus, memory, processor, and device) and the *binding* of software onto execution platform. Each model element (software or execution platform) must be defined by a type and comes with at least one implementation.

5.1.2 AADL application software components

Sequential executions are modeled with so-called subprograms, which can be called from threads and from other subprograms. A thread represents a sequential flow of control that executes instructions. A thread models a schedulable unit that transitions between various scheduling states. A thread always executes within a process. A process represents a virtual address space. Process and threads communicate through typed ports (see Section 5.1.5).

Type and implementation declarations provide a set of properties that characterizes model elements, like the nature and type of the ports. For threads, AADL standard properties include the dispatch protocol (periodic, aperiodic, sporadic, background), the period (if the dispatch protocol is periodic or sporadic), the deadline, the minimum and maximum execution times, along with many others.

In Figure 5.1, p1 and p2 are two processes, t1, t2 and t3 are threads, and step1, step2, step3 are subprograms.

5.1.3 AADL execution platform components

There are four categories of execution platform components in AADL: processor, device, memory and bus.

Processors can execute threads and can contain memory subcomponents. Processors and devices can access memories over buses. Memories represent randomly addressable storage capable of storing binary images in the form of data and code. Buses provide access between processors, devices, and memories. Devices represent entities that interface with the external environment of an application system and may have complex behaviors. A device can interact with application software components through their ports and subprogram features.

In Figure 5.1, CPU1 and CPU2 are two processors, Ds and Da are two (aperiodic) devices, and db1 is a bus.

5.1.4 AADL flows

AADL end-to-end flows identify a data-stream from sensors to the external environment (actuators).

In Figure 5.1, the flow starts from a sensor (Ds) and sinks in an actuator (Da) through two process instances. The first process executes the first two threads while the last thread is executed by the second process. The two devices are part of the execution platform and communicate via a bus (db1) with two processors (cpu1 and cpu2), which host the three threads with several possible bindings. All processes are executed by either the same processor, or any other combination. One possible

binding is illustrated by the dashed arrows. The component declarations and implementations are not shown. Several configurations deriving from this example are modeled with MARTE and discussed in Section 5.3.

5.1.5 AADL ports

There are three kinds of ports: *data*, *event* and *event-data*. Data ports are for data transmissions without queueing. Connections between data ports are either *immediate* or *delayed*. Event ports are for queued communications. The queue size may induce transfer delays that must be taken into account when performing latency analysis. Event data ports are for message transmission with queueing. Here again the queue size may induce transfer delays. In our example, all components have data ports represented as a filled triangle. We have omitted the ports of the processes since they are required to be of the same type than the connected port declared within the thread declaration and are therefore redundant.

5.2 From AADL to UML Marte

5.2.1 Two layers or more

AADL considers two families of components: software and execution platform. However, when specifying flows, a mix is required. In Figure 5.1, the flow starts from a device (execution platform), goes through several processes and threads (application) and ends in another device (execution platform). This domain-specific two-layer approach can be generalized by abstracting the flow itself from the actual (software or hardware) execution platform used. This particular example can be modeled with a 3-layer model (Fig. 5.2), where the top-most layer is the algorithmic view (the flow of data and events), the second intermediate layer describes the software execution platform used (the set of processes and threads) and the last layer describes the hardware execution platform (processors, devices, buses). In Figure 5.2, we have used UML activities to represent the algorithmic view and structured classifiers for the two bottom

layers. More levels could be considered depending on the particular example. The idea is that one layer focuses on one particular aspects or concerns for the designer. For instance, we could also separate the thread view from the processes. In other examples, we could have one layer for the operating system, another for a middleware and another for a virtual machine running on top of them.

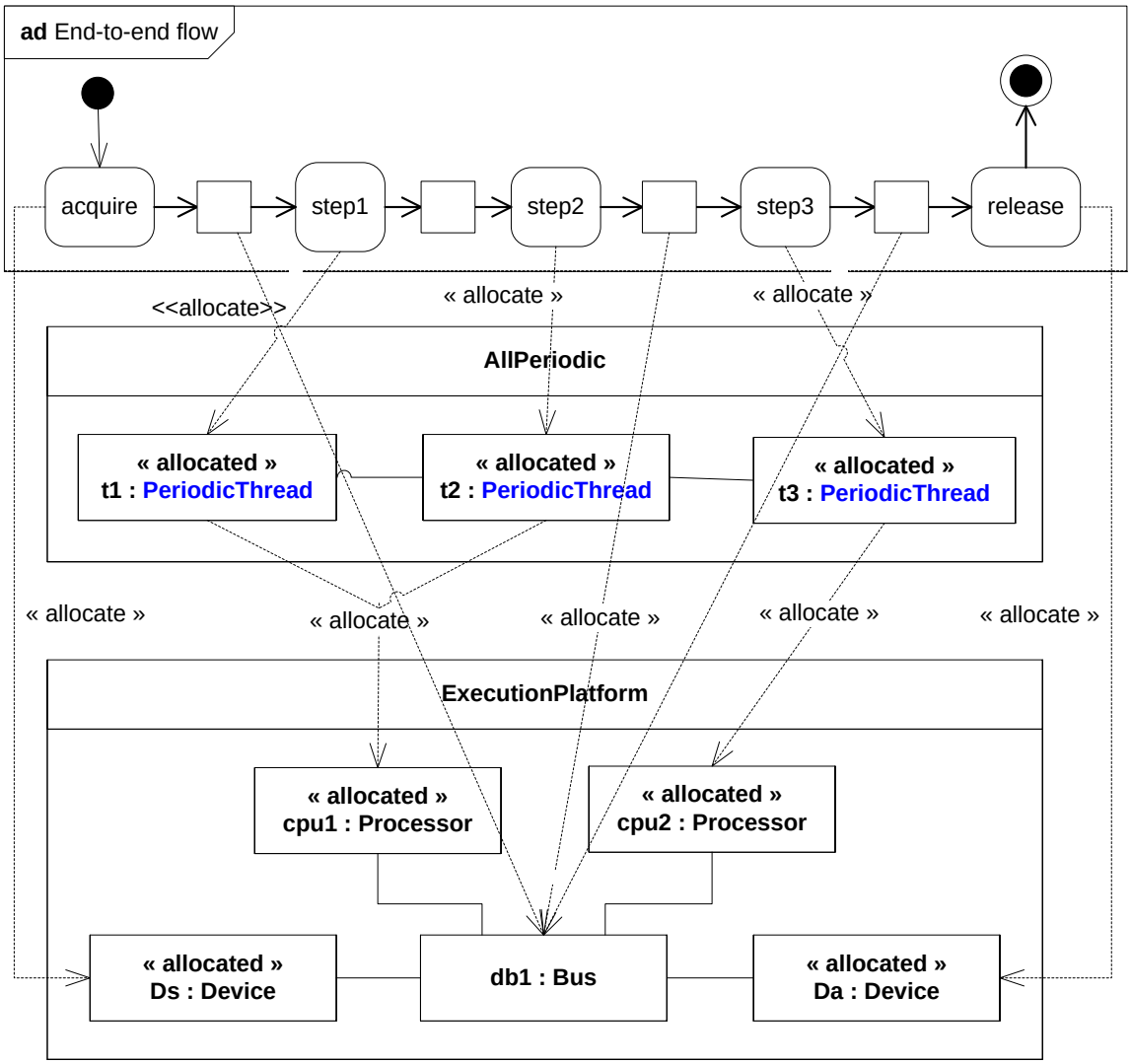

Figure 5.2: Three-layer approach with Marte

In the remainder of this subsection, we discuss possible choices for each layer.

5.2.2 AADL application software components

We have created a UML library to model AADL application software components [LMdS08] (see Fig. 5.3). AADL threads are modeled using the stereotype SwSchedulableResource from the MARTE Software Resource Modeling sub-profile. Its meta-attributes deadlineElements and periodElements explicitly identify the actual properties used to represent the deadline and the period. Using a meta-attribute of type Property avoids a premature choice of the type of such properties. This makes it easier for the transformation tools to be language and domain independent. In our library, MARTE type NFP_Duration is used as an equivalent for AADL type Time.

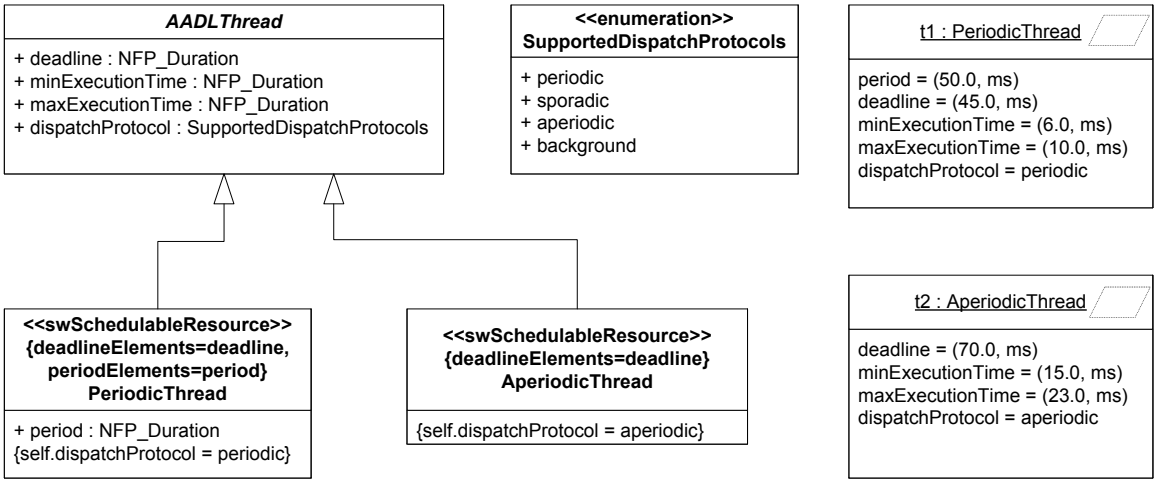

Figure 5.3: A UML/Marte library for AADL threads

This UML/MARTE library mimics the AADL way of building specific types to denote periodic and aperiodic threads. This, as in AADL, demands to change the model when the application software components change. For instance, if $t2$ becomes aperiodic, then we can replace the middle layer by another layer where $t2$ is of type AperiodicThread. Section 5.3 proposes an alternative model that focuses on thread activations and represents them as CCSL clocks.

5.2.3 Modeling ports

UML components are linked together through ports and connectors. No queues are specifically associated with connectors. The queueing policy is better represented on a UML activity diagram that models the algorithm. A UML activity is the specification of a parameterized behavior as the coordinated sequencing of actions determined by *token flows*. A token carries an object, datum, or locus of control. A token is stored in an activity *node* and can move to another node through an *edge*. Nodes and edges have flow rules that define their semantics. In UML, an *object node* (a special activity node) can contain 0 or many tokens. The number of tokens is bounded according to its property upperBound. The order in which the tokens flow out of an object node is selected by setting its property ordering. FIFO (First-In First-Out) is the default ordering value. So, we propose to use object nodes to represent both event and event-data AADL communication links. The token flow represents the communication itself. The standard rule is that only a single token can be chosen at a time. This is fully compatible with the AADL dequeue protocol OneItem. Representing the AADL dequeue protocol AllItems requires to set *edge weights*. This allows any number of tokens to pass along the edge, in groups. The weight attribute specifies the minimum number of tokens that must traverse the edge. Setting this attribute to the unlimited weight (denoted '*') means that *all* the tokens at the source are offered to the target in one single transaction. AADL data ports are modeled with «datastore» object nodes. In such nodes, tokens are never consumed thus allowing multiple readings of the same token. Therefore, AADL data ports are equivalent to a UML data store object node with an upper bound equal to one.

5.3 Describing AADL models with Marte

5.3.1 AADL flows with Marte

We choose to represent the AADL flows using a UML activity diagram. Fig. 5.4 gives the activity diagram equivalent to the AADL example described in Fig. 5.1. The

diagram was built with Papyrus (`http://www.papyrusuml.org`), an open-source UML graphical editor.

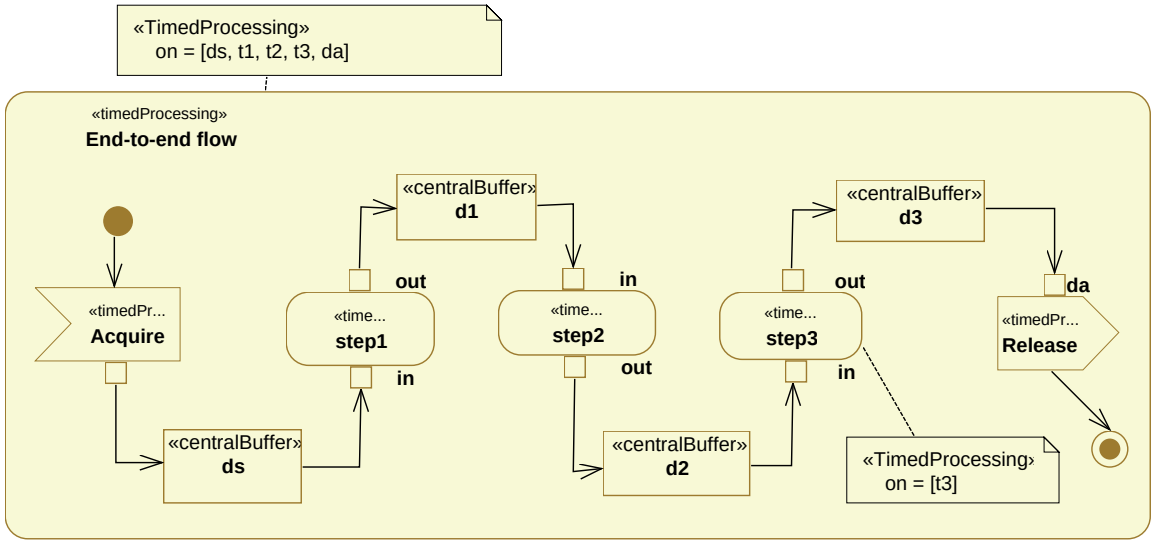

Figure 5.4: End to end flows with UML Marte

As discussed previously, object nodes are used to represent the queues between two tasks. This UML diagram is *untimed* and we use MARTE Time Profile to add time information. This diagram is *a priori* polychronous since each AADL task is independent of the other tasks. CCSL focuses on event occurrences. In this example, we focus on thread activations and build a dedicated, logical and discrete, clock type to represent AADL thread activations (*e.g.,* AADLTask as in Fig. 5.5). Still using UML structured classifiers, we can build another generic layer (see Fig. 5.5) that represents the software execution platform made of three threads, whose activations are modeled as CCSL clocks. The three threads become three logical clocks ($t1$, $t2$, $t3$). Two other logical clocks (ds, da) denote the activations related to the devices. The actual execution semantics as well as the selected dispatch protocol (periodic, aperiodic) are specified as CCSL constraints on these five logical clock constraints.

The tight relationship between the actions in Figure 5.4 and the execution platform is made concrete by a MARTE allocation as shown in Figure 5.2. In that particular case, the stereotype «TimedProcessing» completes the information given by the allocation and states that action *step*1 can only start when thread $t1$ is active.

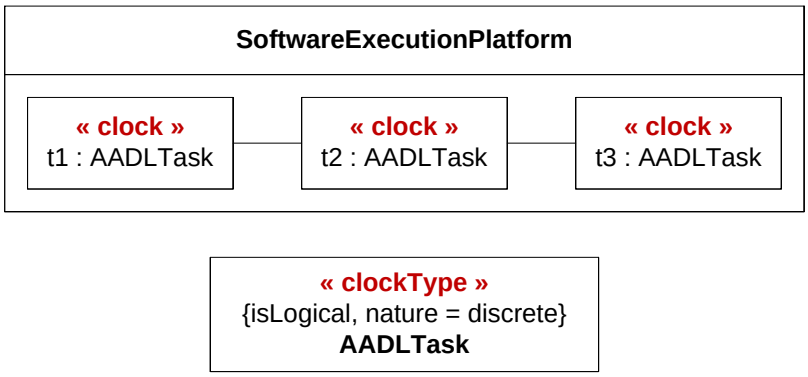

Figure 5.5: AADL thread activation conditions denoted as CCSL clocks

5.3.2 Five aperiodic tasks

The five clocks are *a priori* independent. The dispatch protocols of the tasks determines the CCSL constraints to use. *Aperiodic* tasks (*e.g.,* devices) can start their execution as soon as the data is available on their input port. CCSL relation *alternation* ($\boxed{\sim}$) models asynchronous communications. For instance, action Release starts when the data from Step3 is available in d3. $t3$ is the clock associated with Step3 and da is the clock associated with Release. The asynchronous communication is therefore represented as follows: $t3 \boxed{\sim} da$. Fig. 5.6 represents the execution proposed by TimeSquare with only aperiodic tasks: $ds \boxed{\sim} t1$, $t1 \boxed{\sim} t2$, $t2 \boxed{\sim} t3$, $t3 \boxed{\sim} da$. The optional dashed arrows represent instant precedence relations .

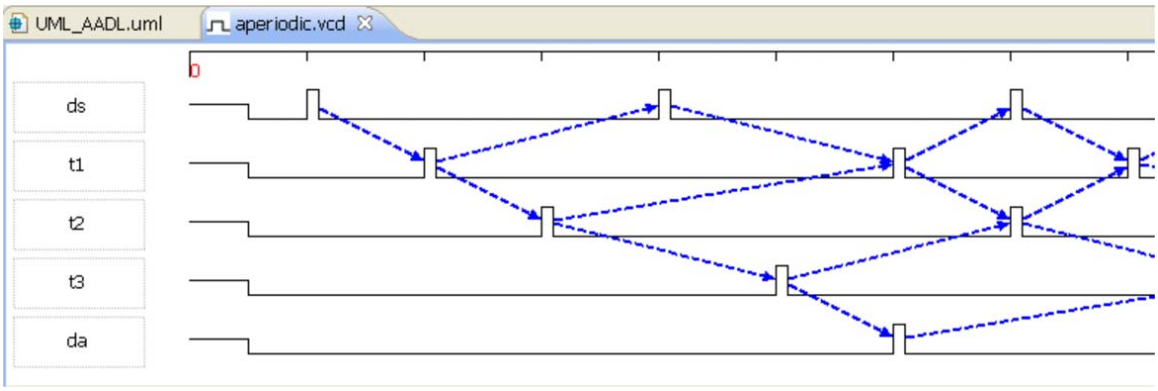

Figure 5.6: Five aperiodic tasks.

This is only an abstraction of the behavior where task durations are neglected. Additionally, note that this specification allows a pipelined behavior: ds occurs a second time before the first occurrence of da. This is because $\boxed{\sim}$ is not transitive. An additional constraint ($ds \boxed{\sim} da$) would be required to ensure the atomic execution of the whole activity. Finally, this *run* is one possible behavior and certainly not the only one. Most of the time, and as in this case, clock constraints only impose a partial ordering on the instants of the clocks. Applying a simulation policy reduces the set of possible solutions. The one applied here is the *random policy* that relies on a pseudo-random number generator. Consequently, the result is not deterministic, but the same simulation can be played again by restoring the generator *seed*.

5.3.3 Mixing periodic and aperiodic tasks

Logical clocks are infinite sets of instants but we do not assume any periodicity, *i.e.,* the distance between successive instants is not relevant. The clock constraint isPeriodicOn allows the creation of a periodic clock from another one. This is a more general notion of periodicity than the general acceptation. A clock $c1$ is said to be periodic on another clock $c2$ with period P if $c1$ ticks every P^{th} ticks of $c2$. In CCSL, this is expressed as follows: $c1$ isPeriodicOn $c2$ period P offset δ.

To build a periodic clock with the usual meaning, the base clock must refer to the physical time, *i.e.,* it must be a chronometric clock. As in Section 4, we can discretize idealClk for that purpose and build c_{100}, a 100 Hz clock (Eq. 5.1).

$$c_{100} \boxed{=} idealClk \text{ discretizedBy } 0.01 \tag{5.1}$$

Figure 5.7 illustrates an execution of the same application when the threads $t1$ and $t3$ are periodic. $t1$ and $t3$ are harmonic and $t3$ is twice as slow as $t1$ (see Eqs. 5.2–5.3).

$$t1 \text{ isPeriodicOn } c_{100} \text{ period } 2 \tag{5.2}$$

$$t3 \text{ isPeriodicOn } t1 \text{ period } 2 \tag{5.3}$$

Coincidence instant relations imposed by the specification are shown with vertical edges with a diamond on one end. Depending on the simulation policy there may also be some opportunistic coincidences. Clock *ds* is not shown at all in this figure since it is completely independent from other clocks.

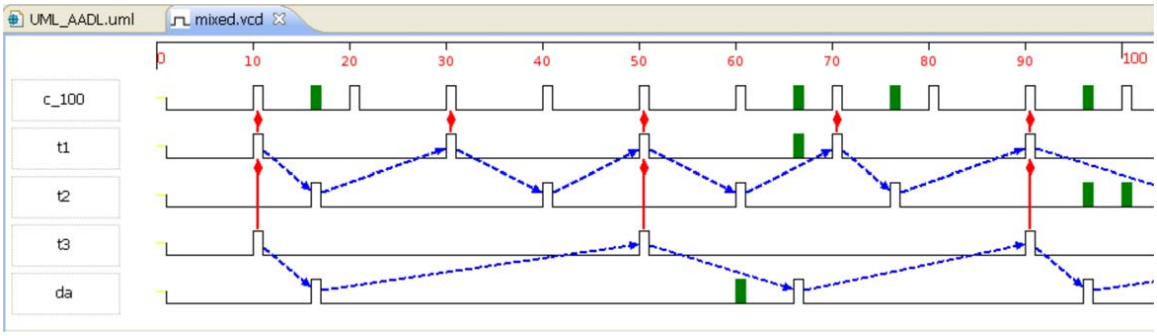

Figure 5.7: Mixing periodic and aperiodic tasks.

Note that, the first execution of $t3$ is synchronous with the first execution of $t1$ and occurs before the first execution of $t2$. Hence, the task *step*3 has no data to consume. This is compatible with the UML semantics only when using data stores. The data stores are non-depleting so if we assume an initialization step to put one data in each data store, the data store can be read several times without any other writings. The execution is allowed, but the result may be difficult to anticipate and the same data will be read several times. When the task $t1$ is slower than $t3$, *i.e.,* when oversampling, some data may be lost. A discussion on these aspects is available in another work [MdS09].

The VCD produced by TimeSquare is annotated with information derived from the CCSL specification to facilitate the interpretation. We have already discussed the instant relations (dashed arrows and vertical edges). Fig. 5.7 also exhibits *ghost*-tick feature. Ghosts may be hidden or shown at will and represent instants when the clock was enabled but not fired. For instance, the first ghost of *c_*100 shows *c_*100 was enabled at the first of occurrence of $t2$, but was not fired. It also shows that even though the second occurrence of $t2$ is simultaneous with the fourth occurrence of *c_*100, it was not strictly imposed by the specification. Additionally, that particular specification happens to be *conflict-free* but we do not have any criterion to decide

on the *conflict-freeness* of a CCSL specification in general: the firing of one clock may disable others. These are classical problems occurring when modeling with Petri nets and that also appear with CCSL because of this precedence instant relation.

5.4 Perspectives

More and more model transformations are proposed from AADL to other (formal) languages [JHR+07, YTG08, CRBS08]. This is part of a larger effort to build platforms (OpenEmbeDD[1], TopCaseD[2]) that rely on largely accepted modeling languages (like AADL, UML, SysML) as front-ends and propose a variety of back-ends to perform various kinds of analyses (performance, schedulability, model-checking ...). The expected result is to enlarge the potential community of users by combining complementary analysis tools and replacing the *niche* formats by graphical modeling languages endorsed by standardization bodies.

However, each of these transformations gives its own semantics to AADL without any guarantee whatsoever that two different transformations actually maintain the same semantics. To address this issue, some environments like TopCased promote the use of a pivot language (like FIACRE) as a common base for the transformation. Our proposed encoding in CCSL can be seen as yet another transformation. Nevertheless, our proposition departs from the others because CCSL is a modeling language that can be directly attached to UML or SysML model elements, through MARTE stereotypes. Therefore, the selected semantics, described in CCSL, becomes explicit within the model instead of being defined by a transformation to an external language. CCSL can then be used as a reference semantic models to guide transformations to other languages. Ensuring that two transformations are equivalent amounts to showing that the result is equivalent to the same CCSL specification. Comparing CCSL with other formal languages is an on-going work. Some preliminary comparison results are given in Chapter 3.

[1] `http://openembedd.org`
[2] `http://www.topcased.org`

Part III

Verification

This part proposes to check existing code against a CCSL specification. The main idea is that logical time is flexible enough to capture causal and time requirements at all modeling levels from the gates (Register Transfer Level - RTL) to the functional level (communicating processes following Cai & Gajski's terminology [CG03]), including the transaction level (TLM). Indeed, synchronous languages [Hal92, BCE+03] have been successfully used at the functional level in safety-critical systems but also as modeling languages able to synthesize optimized RTL code (VHDL) as well as TLM code (SystemC). We discuss here a verification technique in which observers are generated by a structural transformation of a CCSL specification. We have addressed two specific target languages (Esterel [Ber00] and VHDL) that rely on two different programming paradigms, namely synchronous reactive and discrete event. The proposed generation technique can be extended straightforwardly to other languages relying on the same paradigms. Some efforts are currently undertaken to generate code for Scade, another synchronous language, and SystemC, another discrete-event simulation language.

Chapter 7 summarizes the main results presented at LCTES'09 [AMallet09]. A research report [And10] details a library of Esterel modules that allows the automated construction of observers for any CCSL specification. The key of the success resides in the closeness of the semantics of CCSL and Esterel that are both instant-based fixed point semantics. Chapter 8 explores an adaption of this process for VHDL implementations. The adaptation is not trivial because of the semantic gap between the CCSL semantics and the simulation semantics of VHDL based on microsteps with delta cycles. Building a library of VHDL components for CCSL specification checking requires a careful comparison between the two semantics. This work was initiated during the PhD thesis of Aamir Mehmood Khan, who defended in March 2010 [Meh10]. A position paper was presented at FDL 2009 [MKMAdS09]. A more thorough analysis has been recently published [AMD10]. General aspects regarding the construction of observers for CCSL specifications, not specific to either VHDL or Esterel, are briefly presented in the remainder of Chapter 6. The proposed generic transformation was implemented in TimeSquare by Antoine Boulinguez during his training period to obtain his second year degree from Nice's institute of technology.

Chapter 6

Building language-specific observers for CCSL

6.1 The generation process

Verification by observers is a technique widely applied to property analysis / checking [HLR94, ABL98, BBKT05, BJB05]. As its name indicates, an observer continuously observes executions of a system to detect some specific, possibly undesirable, behaviors. Often the observers are used at runtime or in simulation. An observer can see input, output and internal events or values of the program. If the observed evolution does not satisfy one expected property, the observer enters a failure state and reports a *violation*. Usually, the observer is written in the same language as the model (*e.g.,* Esterel, SystemC, VHDL) as long as this language supports the parallel composition.

A CCSL specification is a set of possibly inter-related constraints that express *safety properties*. Our goal is to generate an observer from the CCSL specification. We propose to create a library of "components", for each CCSL constraint (relations and expressions) and perform a structural generation. Relations and expressions are of a different nature. An expression defines a new clock. For each expression, we build a component called a *generator*. A relation constrains two clocks. Since we use

non-intrusive observers, we cannot force anything to happen and we can only observe violations. For each relation, we build a component called a *relationObserver*, which has two inputs (one for each clock) and one output: the violation signal. A violation of the specification will occur if any violation output of any relation *relationObserver* is asserted.

The semantics of each CCSL constraint is given by an SOS rule that determines, given a configuration, which clock MUST, CAN or CANNOT tick. The idea is to encode the SOS rule directly in the target language. Clocks are encoded by three-state values: respectively 1, X, 0 in VHDL. A violation occurs either when the clock must tick according to the specification and does not actually tick in the implementation, or when the clock cannot tick (according to the specification) and actually ticks (in the implementation). When a clock can tick (according to the specification), then it may or may not actually tick (in the implementation).

A *generator* creates a new clock from its inputs. Those inputs are fully determined by the program under test or by another generator, and therefore the resulting clock is fully deterministic. For this reason, the specification of a generator is exactly the SOS rule of the corresponding expression. However, whereas the Esterel encoding of the SOS rule will only provide a constructive solution, a naive encoding in VHDL might cause glitches and false violations. The two following chapters describe the proposed encoding in Esterel and in VHDL.

An *observer* must verify that something bad never happens. However, the SOS rule corresponding to a relation specifies what should happen. Consequently, a violation occurs when the incoming clocks falsify the enabling condition. Therefore, the *violation condition*—checked by the observer—is just the logical negation of the enabling condition. The initial conditions and the internal state evolution rules are directly encoded from the SOS rewrite rules. Here again, naive implementations may cause false violations due to the lack of constructiveness.

From the SOS rules, it is possible to obtain the specification for the generators and the observers. Both of them consider logical clocks as inputs. However, whereas Esterel signal can be seen as CCSL logical clocks with values, the VHDL valued signals

are very different. In both cases, a conversion between signals (or possibly a combination of signals) into logical clocks is required. Such an adaptation is done by specific hand-made components called *adapters*. Adapters are also very useful to use the same specification for different implementations of the same system at different abstraction levels. Adapters are sometimes called *transactors* when they play such a role. Even with a fairly large library of common adapters, it may always appear new cases that have to be designed individually. For instance, in Figure 6.1, the adapter TF is a sequential component that builds the logical clock tb_f, so that it ticks whenever the signal $PSEL$ has been asserted HIGH in two consecutive cycles of signal CLK. This particular example is further discussed in Chapter 8.

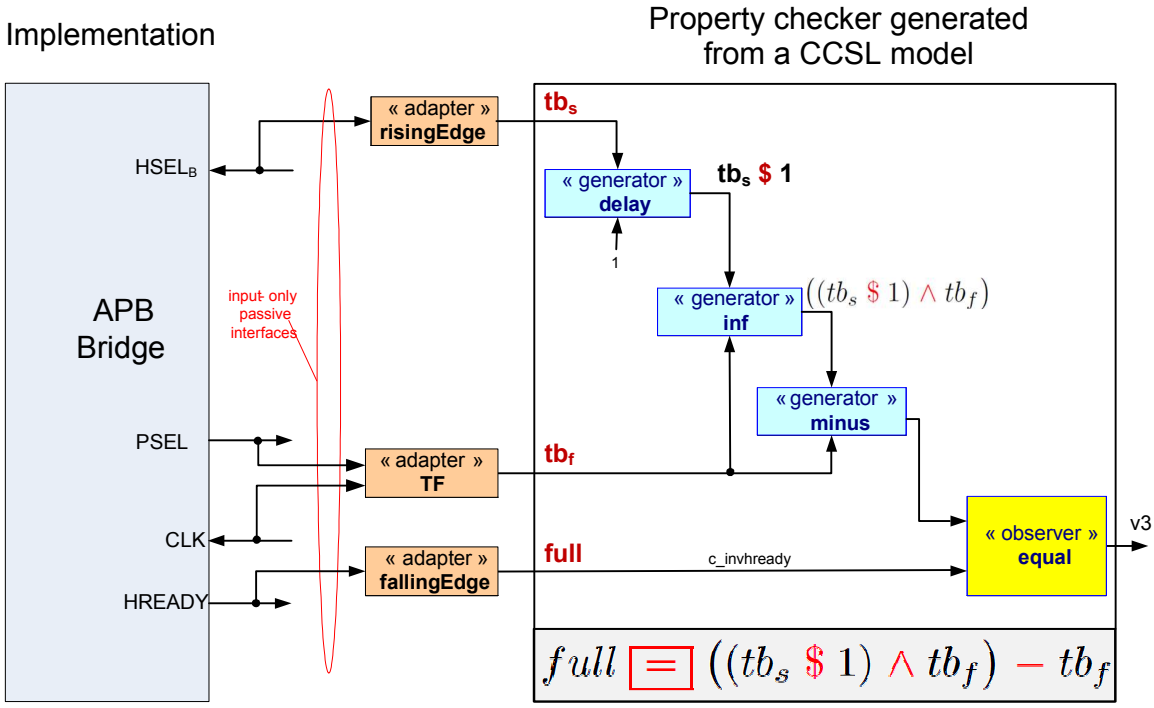

Figure 6.1: The observation network structurally reflects the CCSL model

CCSL models conform to the CCSL metamodel and can therefore be represented as a tree. The root of this tree is the clock relation, the leaves are clocks, and intermediate nodes are clock expressions. Thus, the components used to implement a clock relation checker are assembled as a tree that reflects the same structure. An observer

component is the root, adapter components are the leaves, and generator components are the intermediate nodes. This structure is acyclic: information starting from the leaves eventually arrives at the root. For optimization reasons, some components may be shared. So, the actual structure can be a Directed Acyclic Graph (DAG) of components, whose maximal elements are observer components, and minimal elements are adapter components (see Fig. 6.1). Be it a tree or a DAG, we call an assembly of components used to check a clock relation an *observation network*.

6.2 Adapters

6.2.1 In Esterel

Pure Esterel signals are strictly equivalent to CCSL logical clocks. Valued signals carry a logical clock and a value. Building adapters for Esterel can be as simple as extracting the pure signal from any valued signal. It could also involve a more complex Esterel module that combines different Esterel signals and takes the values into account. For the example of the digital filter discussed in Chapter 7, the former case has been retained in most cases. The following Esterel adapter binds the output valued signal OutPixel to a pure signal c_OutPixel, which stands for a CCSL clock $outPixel$. That is to say, clock $outPixel$ ticks whenever the signal OutPixel is present, whatever its value.

```
sustain c_OutPixel if OutPixel
```

6.2.2 In VHDL

In VHDL, it makes no sense to test for the presence of a signal. So we have to find a way to encode ticks of logical clocks. For this purpose, we use a 'pulsed' signal whose pulses represent the clock ticks. The width of a pulse is ε (`EPSILON`). ε is strictly positive but 'as small as possible', *i.e.,* far smaller than the minimal duration (Δ_{min}) between application events. $\varepsilon > 0$ ensures that the rising and the falling edges of the

pulse occur at different simulation time, *i.e.,* not within another delta cycle at the same simulation time. $\varepsilon \ll \Delta_{\min}$ makes that the pulse falling-edge is the simulation instant immediately following the pulse rising-edge. A pulse is easily generated in VHDL by assigning waveforms to a signal. Execution of c_out <= '1', '0' **after** EPSILON; produces a pulse whose width is EPSILON, a given constant typed Time.

In its simplest form, an adapter takes a single input signal and generates a pulse on a particular (VHDL) event on this signal (*e.g.,* a rising-edge). The library provides adapters for rising-edge and falling-edge. Sometimes, specific adapters must be written. This was the case for example addressed in Chapter 8, which uses an adapter that implies a sequential behavior on two signals.

6.3 Relation observers

To compare the process of building observers in Esterel and in VHDL, we choose the example of the CCSL relation precedes (denoted $\boxed{\prec}$). For a specification, *i.e.,* a set of discrete clocks C, we associate a function χ called *configuration*, $\chi : C \to \mathbb{N}$ that gives the current time, *i.e.,* the index of the current instant for each clock. The initial configuration χ_0 is so that $(\forall c \in C)(\chi_0(c) = 0)$. In the operational semantics of the *strict precedence* ($c_1 \boxed{\prec} c_2$), we compare the configurations for the two clocks c_1 and c_2 (*i.e.,* $\delta = \chi(c_1) - \chi(c_2)$) and we allow c_2 to tick only when $\delta > 0$. This is formally expressed with the following SOS rule:

$$\frac{\delta \triangleq (\chi(c_1) - \chi(c_2))}{[\![c_1 \boxed{\prec} c_2]\!] = (\delta \leq 0 \Rightarrow \neg c_2)} \qquad (\textit{strict precedence})$$

This rule (or rather its negation) must be encoded in the target language. Note that, the rule essentially prevents δ from being negative (this is the case, for the initial configuration). Therefore, the observer should count the occurrences of c_1 and c_2 and should emit a violation whenever δ becomes negative.

6.3.1 In Esterel

The encoding of the SOS rule is straightforward in Esterel:

```
1  module Ccsl_R_precedes:
2    input c1,c2;
3    output Violation;
4
5    signal Delta: value unsigned init 0 in
6      sustain {
7      Violation if (pre(?Delta) = 0) and c2,
8      ?Delta <= pre(?Delta) + 1 if c1 and not c2,
9      ?Delta <= pre(?Delta) - 1 if not c1 and c2
10     }
11   end signal
12 end module
```

Note that this implementation uses an unsigned signal (Delta, line 5) to encode the difference in the number of occurrences between clocks c1 and c2. This makes unbounded models that, consequently, cannot be verified by model-checkers. To run a model-checker, the unsigned variable must be bounded prior to verification.

6.3.2 In VHDL

To avoid false violations due to glitches, we use postponed processes. Since a relation observer is always at the end of the observation network, the code of an observer can be executed at the last delta cycle. The internal state δ is represented by an integer variable *delta*, initialized to 0 (line 4). The negation of the enabling condition (*i.e.,* the violation condition) is $(\delta = 0) \wedge c2$. Lines 11 to 14 check this condition and set `violation` accordingly. Lines 15 to 20 maintain the internal state. The whole process is postponed (line 8), so that its code is executed only when all signals are stable.

```
1  architecture Ccsl_R_precedes of Ccsl_R_Observer is
2  begin
3    postponed process(c1,c2)
4      variable delta:integer := 0;
5    begin
6      if (delta = 0) and (c2 = '1')
7      then violation <= '1'; -- violation
8      else violation <= '0';  end if;
9      if c1 = '1' then delta := delta + 1; end if;
10     if c2 = '1' then delta := delta - 1; end if;
11   end process;
12 end architecture Ccsl_R_precedes;
```

6.4 Generators

Generators implement CCSL expressions, which build new clocks from existing ones. Whereas the semantics of a CCSL relation is purely combinatorial, the semantics of expressions is sequential. Consequently, some SOS rules decide whether the constructed clock ticks or not, for a given configuration. Other SOS rules rewrite the expression into a new one. As an example, we study expression *wait*. wait 5 c is an expression that ticks once at the 5^{th} occurrence of c. It departs from the *delay* operator by not being reentrant. It has been chosen because of its simplicity. Encoding the reentrant delay is a bit more complex but follows the same line.

$$\frac{\beta \triangleq (n = 0)}{[\![\text{wait } n\ c]\!] = (\beta \wedge \mathrm{c})} \qquad (wait)$$

$$\frac{c \in F \qquad n > 0}{\text{wait } n\ c \rightarrow \text{wait } (n-1)\ c} \qquad (RWwait)$$

Rule *(wait)* states that the clock built by wait n c only ticks when c ticks and $n = 0$. The second rule decrements the counter n when c ticks. If c does not tick, nothing happens. When n reaches 0, the new clock ticks whenever c ticks.

6.4.1 In Esterel

A direct Esterel encoding of the rules is as follows

```
1  module Ccsl_E_wait:
2    input c;
3    constant N: unsigned;
4    output o;

6    signal count : unsigned init = N do
7      every c do emit o if ?count=0 end every
8    ||
9      sustain {
10     ?counter <= pre(?count) - 1 if c and ?count>0
11     }
12   end signal
13 end module
```

The values of the constant N (line 3) is set at the *compile-time* module instantiation. We use a local unsigned variable *count* (line 6) to count up to N occurrences of c. Line 7 implements the first rule (*wait*), whereas lines 9-10 implement the second rule (*RWwait*). The parallel operator ($\|$) put them together. However, let us note that the expression *wait* is a primitive operator in Esterel. The above module is equivalent to wait N c; **emit** o; but the purpose of this example is to illustrate the possible automatic encoding from the rewriting rules.

6.4.2 In VHDL

Since a generator is not a maximal element in the observation network, it cannot be implemented as a postponed process. The idea is to realize it as two separate processes. The first process, named `surface`, deals with the combinatorial behavior. This process drives the generator output `o` and may introduce glitches. The second process, named `depth`, is sequential and manages the internal state. `depth` is a postponed process, thus it works only when the observation network has stabilized. The names 'surface' and 'depth' come from the synchronous language compilers that also separate combinatorial and sequential evolutions. This is illustrated on a generator that implements *delay*. Such a generator has one input clock `c`, a natural number input parameter `n`, and one output clock `o`.

```
entity Ccsl_E_delay is
  generic (N: NATURAL := 1);
  port (c: in  bit; o: out bit:= '0');
end Ccsl_E_delay;

architecture  Ccsl_E_delay_arch of Ccsl_E_delay is
  signal count : NATURAL := N;
begin
  surface: o <= '1' when c='1' and count=0 else '0';

  depth: postponed process (c)
  begin
    if c='1' and count>0 then count := count - 1;  end if;
  end process;
end Ccsl_E_delay_arch;
```

The local counting signal `count` is declared and initialized at line 6. This signal is accessible by the two processes: `surface` for reading, and for reading and writing. Line

8 implements the enabling condition (`surface`). Process `depth` updates the counting signal whenever c ticks and until it reaches 0. Note that, the output o is also a pulsed-signal if c is a pulsed-signal. For all expressions and relations, it is important to check that this property is preserved on all signals that represent logical clocks.

6.5 Perspectives

By providing this observer-based verification process, we extend the possible use of CCSL in a design flow. This proposed flow is as follows. It starts with a UML model. The model is annotated with CCSL constraints by applying the profile MARTE and using its stereotypes. The resulting executable specification is assessed and refined by using feedbacks from TimeSquare simulation. Finally, the implementation is validated through observers generated from the CCSL specification. Following this flow, the CCSL specification can act as a *golden model* against which, several alternative implementations, possibly at different abstraction levels, can be checked. For now, this process is applicable with two target implementation languages (Esterel and VHDL). Extensions to similar languages (Scade, SystemC) are ongoing. Relying on Esterel gives access to its formal verification suite, thus extending TimeSquare capabilities.

Chapter 7 illustrates the approach with Esterel and discusses the use of Esterel Studio verification suite. Chapter 8 illustrates the process to verify an AMBA AHB to APB bridge.

Chapter 7

Verifying Esterel implementations

In this chapter, we use the simple example of a digital filtering video application to illustrate our process applied to the Esterel language. The example is described in Section 7.1. Section 7.2 uses CCSL to specify the expected behavior of the digital filter. The specification is simulated with TimeSquare, an environment we have developed, dedicated to MARTE Time Specification and CCSL analysis. We then rely on Esterel Studio formal verification facilities to check the conformance of a candidate Esterel/SyncCharts implementation with its specification.

7.1 Example: a digital filter

This section introduces the example selected to illustrate our proposal: a simple digital image filtering (DF) application. This example is borrowed from the "*Getting Started Manual*" of Esterel Studio and was designed as a tutorial on its modeling capabilities.

DF is used in a video system. It reads groups of pixels from a memory, filters them and sends output pixels out to a display device.

One image is composed of LPI lines, each line consists of PPL pixels. The pixels are stored in words. A word contains PPW pixels, a line WPL words $\left(WPL =\right.$

$\lceil PPL/PPW \rceil$). The pixel transformation (digital filtering) is defined by a dot product:

$$y[k] = \sum_{j=-L}^{j=+L} c[j] * x[k-j] \tag{7.1}$$

where k is a natural number, index of the pixels in a line, y is an array of output pixels, x is an array of input pixels, and c is an array of $2L+1$ constant coefficients.

DF has four signal ports. The input port InWord conveys WORD values, the output port OutPixel conveys PIXEL values. The two other output ports (Ready and EndOfLine) are *pure signals*, that is, they do not carry values and are used for signaling event occurrences. A rough specification of the behavior of DF is as follows. DF requests a new incoming word by asserting Ready ①. In response, an external memory sends back the next word of the image (signal InWord). OutPixel are sequentially issued after receiving InWord ② and performing the filtering. EndOfLine is asserted each time the last pixel of a line is emitted ③. The circled numbers (①, ②, ③), in Figure 7.1, refer to instant relations and are discussed in the next subsections.

7.2 CCSL specification

Events of the system are modeled as logical clocks and the specification is imposed by applying constraints to these clocks. An event can be a signal receipt (e.g., $inWord$), a signal emission (*e.g.,* $outPixel$), or the presence of a pure signal (*e.g.,* $ready$, $endOfLine$). A logical clock ticks each time the associated event occurs. For convenience, we denote the clock associated with a signal by the name of the signal in italic and with an initial lower case letter.

Precedence arrows and coincidence edges in Figure 7.1 represent some instant relations implied by the specification. Precedence relation ① states that for each request (each tick of $ready$) a new word must be released ($inWord$ must tick). Precedence ② expresses that each received word produces four output pixels. The rounded-corner rectangle makes it explicit that a word gives rise to four output pixels exactly. Coincidence ③ says (for the unlikely case of a 8-pixel line) that the first tick of $endOfLine$

is coincident with the 8^{th} outpixel.

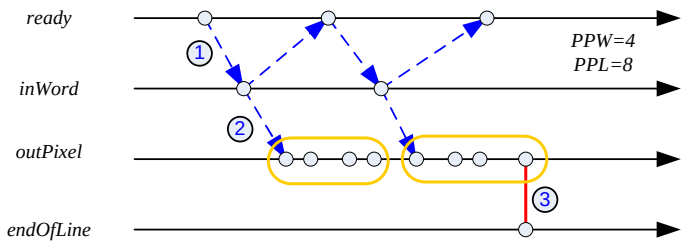

Figure 7.1: Some time constraints on the DF behavior

Of course, the instant relations represented in Figure 7.1 hold for all lines of the image and all parameter settings. Instead of expressing instant relations on an instant-pair base, it is more convenient to apply constraints on clocks directly. Adequate clock constraints that correctly implement the specification are informally described in the following subsection.

Clock constraints are a generic way to define various aspects of a specification. They may derive from the algorithm itself or from performance requirements, and also from the data structure used or the operating mode. We have identified here four primary constraints covering several of these aspects.

(Cstr ①) The specified *protocol* implies that each request ($ready$) is followed by a new word ($inWord$) and that no new request is sent before the preceding request has been acknowledged. This is an *alternation* constraint where, $ready$ alternates with $inWord$ ($ready \boxed{\sim} inWord$). In terms of clocks, each instant of $ready$ precedes an instant of $inWord$, which precedes the next instant of $ready$, and so on.

(Cstr ②) Because of the chosen *data structure*, input pixels are packed within words of length PPW, whereas output pixels are individually released. The algorithm imposes that the number of pixels is preserved. A by-packet precedence relation denotes such a fact. Each tick of $inWord$ precedes a group of PPW consecutive ticks of $outPixel$: $inWord \boxed{\prec} \Big(outPixel/PPW\Big)$.

(Cstr ③) $endOfLine$ ticks every PPL ticks of $outPixel$. This constraint directly reflects the semantics of the end of line: $endOfLine \boxed{=} outPixel \blacktriangledown (0^{PPL-1}.1)^{\omega}$.

(Cstr ④-⑤) Additional non-functional constraints must be set to impose readiness and reduce communication buffers. Such a constraint should avoid delaying unnecessarily the processing of received input words and gives rise to further precedence constraints between $outPixel$ and $inWord$.

The periodic pattern (in ③) models regular data flows. Here, each pixel line has the same length, and the same transformation periodically applies to each line.

The pixel transformation being a dot product (Eq. 7.1), each output pixel depends on $2L + 1$ consecutive input pixels. CCSL only deals with the structural relations, therefore the actual transformation is not relevant and the data dependencies (between inputs and outputs of the pixel transformation) are implemented by several precedence constraints that are surely much stronger than (Cstr ②)). Figure 7.2 shows these precedence constraints for one single image row in the simplistic case where $PPL = 8$, and $L = 2$. A more general characterization is given in [AMallet09].

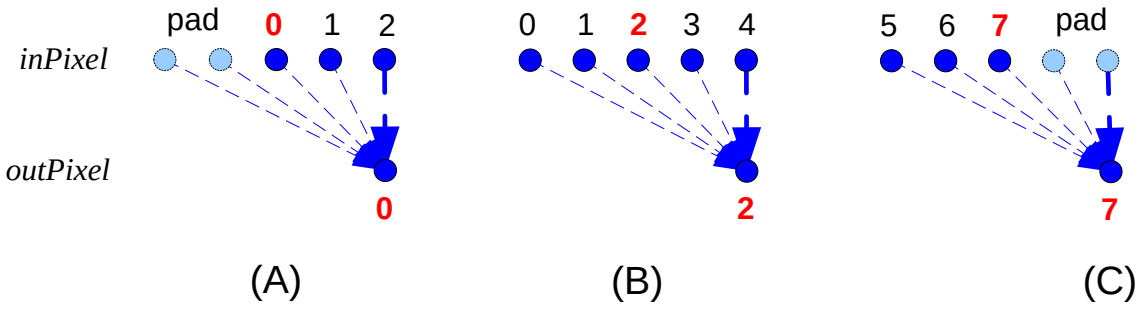

Figure 7.2: Pixel dependency

As always in pipelined specifications, three phases must be considered for each line. The prolog, when filling the pipeline, the kernel, when the pipeline is in a steady state, the epilog, when draining the pipeline.

Figure 7.2 (A) shows the beginning of the line processing (prolog) where padding pixels are necessary to apply the dot product. OutPixel[0] depends on InPixel[-2..2]. A default value is given to padding pixels InPixel[-2] and InPixel[-1].

Figure 7.2 (B) illustrates the steady phase (kernel) where the computation of each output pixel depends on five inputs pixels. OutPixel[2] depends on InPixel[0..4] ... InPixel[4]. Because of the implicit ordering on input pixels (InPixel[j] precedes InPixel[k], for any $j < k$), only one precedence is required: InPixel[4] must precede OutPixel[2].

Figure 7.2 (C) represents the ending of the 8-pixel line processing (epilog). OutPixel[7] depends on InPixel[5..9]. InPixel[8] and InPixel[9] are also padding pixels for which a default value is assumed.

Since signal InPixel is not part of the interface, the precedence relations between InPixel and OutPixel have to be expressed as precedence relations between InWord and OutPixel (Constraint ④). Relation ⑤ is a back-pressure constraint to guarantee that output pixels are delivered fast enough so the communication buffer contains at most two words.

$$\left(inWord \blacktriangledown (0.1)^{\omega}\right) \boxed{\prec} \left(outPixel \blacktriangledown 0^2.\left(1.0^7\right)^{\omega}\right) \qquad ④$$

$$\left(outPixel \blacktriangledown 0.\left(1.0^7\right)^{\omega}\right) \boxed{\prec} \left(inWord \blacktriangledown (0.1)^{\omega}\right) \qquad ⑤$$

Overall, the specification mixes synchronous (Cstr ③) and asynchronous (Cstr ①, ②, ④) constraints and involves functional and non-functional aspects. Such a specification is a good example to have a broad overview of CCSL expressiveness.

7.3 Running simulations with TimeSquare

Figure 7.3 illustrates a correct run for the given specification generated by the TimeSquare simulation engine. Note that alternative runs may also be correct since the simulation engine generates one possible solution.

TimeSquare VCD viewer displays instant relations when requested. Precedence relations are displayed as dashed arrows. Coincidence relations are shown as vertical lines with a diamond on the side of the super clock. When packet-based constraints (as in ②) are used, the packets are depicted as rounded-corner rectangles surrounding

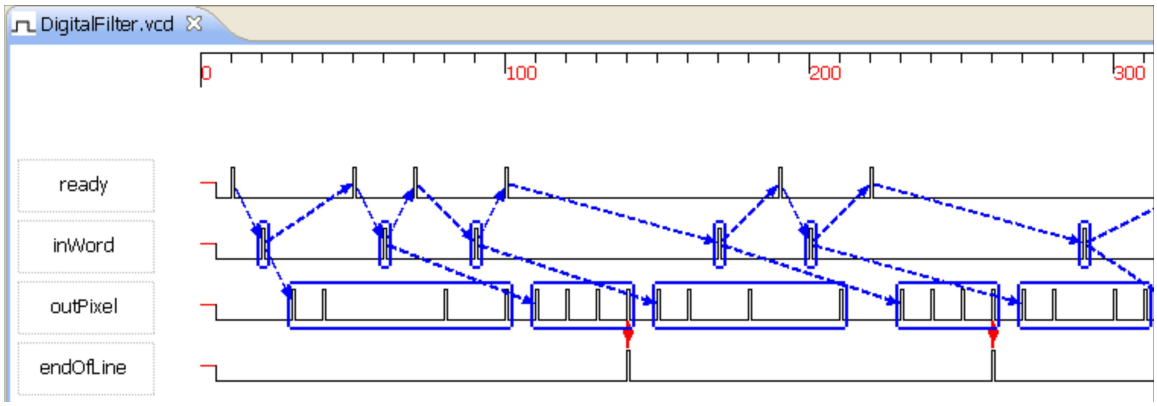

Figure 7.3: One acceptable solution generated by TimeSquare

the related clock ticks.

Even though simulation can help to discover some specification inconsistencies, it only considers one possible solution at a time. It must be combined with exhaustive analysis for corner bug detection and formal verification of safety requirements. This is addressed in Section 7.4.

7.4 Analysis with Esterel observers

Now that we are confident that our CCSL specification includes all the properties that we want to verify, we want to use this specification on an existing implementation. We took the code provided in the previously mentioned Esterel-Studio's Getting Started manual. The DF program consists of two parts: the Feeder, written in SyncCharts [And96], and the Filter, written in Esterel [Ber00]. Esterel and SyncCharts semantics are fully compatible, and any SyncChart can be translated into a semantically equivalent Esterel code [And04, Zaf05].

The Esterel compiler is part of a comprehensive development environment named Esterel Studio. This environment provides compilation, simulation, coverage, verification and code generation facilities. In this subsection we consider only the fourth one. Formal verification of Esterel programs relies on two complementary technologies: 1) Symbolic model checking based on a BDD technology, 2) Bounded and Full

model checking based on SAT-technology. *Bounded Model Checking* (BMC) is efficient in searching for bugs in design and property specifications. Since BMC can only *falsify* properties, it cannot be used to prove a property correct. On the contrary, *Full Model Checking* (FMC) can prove that a property holds, but the process may take a great amount of time. FMC makes its best to combine SAT-solver with induction [SSS00] and improved strategy combining interpolation and SAT-based model checking [McM03]. Symbolic Model Checking (SMC) can be used both to falsify and to prove properties. The drawback of this BDD-based model checking is the possibility to run out of memory and thus be inconclusive.

A property to check is directly expressed in Esterel either as an *assertion* or as an *observer*. An assertion may represent an *assumption* about the execution environment of the program to check. An assertion also allows implementing parts of its intended behavior as executable and verifiable predicates, into the design code. An observer is a special program unit, not part of the design, and used in property checking. It continuously observes input and output signals of the program and detects possible property violations. Used in combination with model checking, observers are a powerful means to find bugs and formally establish properties. If a violation occurs, the model checker generates a simulation trace leading to this violation, thus exhibiting a counter-example of the checked property. Note that the observers are non-intrusive: they do not alter the behavior of the tested Esterel program.

Generally, verification starts with a search for bugs or property violations. This is done by BMC. In the application at hand, a violation is detected when checking Cstr ①. The model checker generates a counter-example sequence of 13 reactions. This trace confirms the presence, at instant 12, of a spurious signal Ready. This unexpected emission of Ready is caused by a abnormal use of a weak preemption. This abnormal behavior is corrected by forbidding the emission of Ready when processing the last input word of a line. With the modified program all the CCSL constraints are satisfied by applying FMC.

Chapter 8

Verifying VHDL implementations

Chapter 6 describes the principle of a component-based implementation of CCSL constraint observers. Here, we apply this principle to VHDL. Information (clock ticks) has to propagate through the observation network before reaching the terminal node (an observer component). In this network, different paths with different lengths can cause glitches because of the microstep semantics. So, a naive implementation might detect *false* violations. The challenge was to devise *delta-delay insensitive* VHDL observers. This chapter describes our solution and illustrates the process on a AHB to APB Bridge that is part of a larger design (a LeonII-based embedded system, whose VHDL model is available in open source[1]). The example is introduced in Section 8.1.

8.1 Example: an AMBA AHB to APB Bridge

The *Advanced Microcontroller Bus Architecture* (AMBA) specification defines an onchip communications standard for designing high-performance embedded microcontrollers. We consider two buses defined with the AMBA specification:

- The *Advance High-performance Bus* (AHB) for high-performance, high clock frequency system modules;

[1] `http://www.gaisler.com`

- The *Advanced Peripheral Bus* (APB) optimized for minimal power consumption and reduced interface complexity to support peripheral functions.

In a typical AMBA architecture, which contains both types of bus, an AHB to APB *bridge* is necessary. The APB bridge interfaces the AHB to the APB and converts system bus transfers into APB transfers. It buffers address, control, and data from the AHB, drives the APB peripherals and returns data or response signals to the AHB. On a data transfer request, it decodes the address using an internal address map and generates a peripheral select, PSELx. Only one select signal can be active during a transfer. The bridge drives the data onto the APB for write transfers or, in case of read transfers, it drives the APB data onto the system bus.

Figure 8.1 illustrates a write transfer on the APB bridge. The transfer starts when the destination address is written in HADDR. A central address decoder is used to provide a select signal, HSELx, for each slave on the AHB bus. The select signal is a combinatorial decode of the high-order address signals. Let $\mathsf{HSEL_B}$ be the select signal for the bridge. When HADDR is set to a value within a given address range, $\mathsf{HSEL_B}$ is set to high and the bridge should initiate a transfer (at T2). A write transfer is initiated when HWRITE is set to high, a read transfer is initiated otherwise.

For write transfers, the data must be given in HWDATA and must be available at the next cycle (at T3). Each transfer takes exactly two cycles to complete on the APB. In a first step (T3-T4), the address is further decoded by the bridge to select the appropriate APB slave. The address is set in PADDR, the date is set in PWDATA and the appropriate PSEL signal is asserted. In a second step (T4-T5), PENABLE is asserted and the write transaction is completed.

8.2 CCSL specification

From this specification we attempt to extract a *higher view* of the transaction and identify the logical events that can be modeled as logical clocks. We identify two logical clocks here: tb_s (transfer bridge start), whose instants characterize the initiation of the transfer; tb_f (transfer bridge finish), which characterizes the completion of the

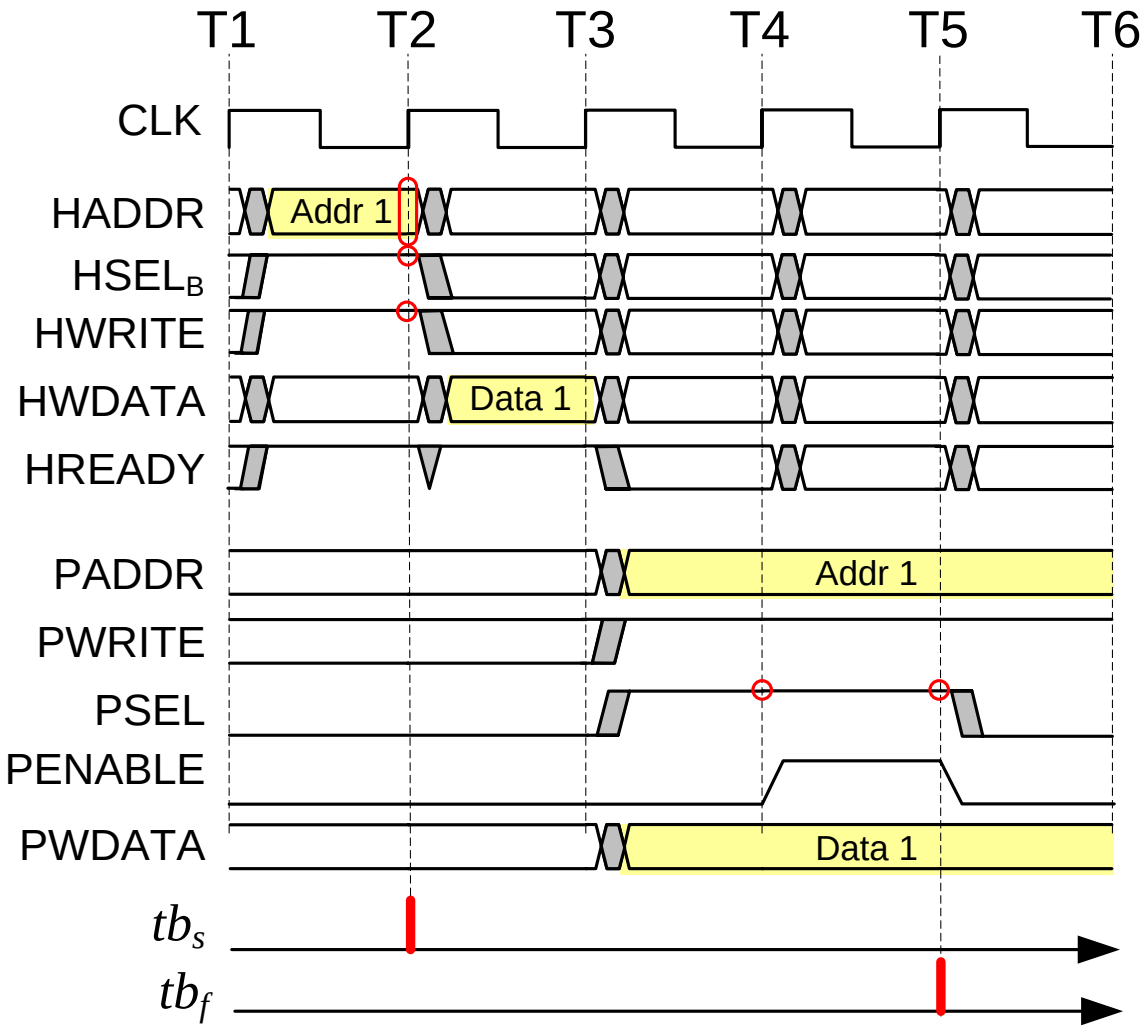

Figure 8.1: A typical write transfer through the bridge

transfer. A basic property that should be valid for any kind of transfer is that the initiation should always precede the completion. Such a property can be expressed in CCSL using the relation *precedence* (see Eq. 8.1).

$$tb_s \prec tb_f \tag{8.1}$$

To bridge the abstraction gap between the high-level logical view of the specification and the specific RTL implementation, we need an *adapter*. To build such an adapter, we have to decide what exactly is considered as the initiation of the transfer and what is considered as the end. Focusing on the initiation (tb_s), several solutions are possible. An asynchronous view (ignoring the bus clock CLK) could consider the rising edge of signal $\mathsf{HSEL_B}$ (*i.e.,* some time between T1 and T2) as the actual initiation. A synchronous view would rather consider the rising edge of signal CLK when $\mathsf{HSEL_B}$ is high (at T2). Both solutions are acceptable. The latter one has been chosen and depicted in Figure 8.1.

Considering now the case of the transfer completion (tb_f), the adaptation appears a bit more complex. A synchronous interpretation of the transfer dictates that the actual completion occurs on the second raising edge of CLK when PSEL has been continuously high during two cycles (at T5). An asynchronous interpretation could consider the transfer completion on the falling edge of PENABLE. We chose the synchronous version.

Bounded transfers The simple specification provided in Eq. 8.1 is general to any request/response or producer/consumer system. In the APB bridge, the request is the transfer initiation (tb_s) and the response is the transfer completion (tb_f). Such a transfer is unbounded, it only specifies that the response must come at some point but it can be arbitrarily far from the request. In most cases, this is not suitable and transfers need to be bounded. This is actually the case for the AHB to APB bridge, whose specification explicitly mention a bound of 2. That is to say that at most two (but no more) consecutive requests can be performed even though the response to the first request has not been given yet.

For a buffer of size n, the specification would be that the k^{th} response always precedes the $(k+n)^{\text{th}}$ request: $(\forall k \in \mathbb{N}^{\star})\ tb_f[k] \prec tb_s[k+n]$. This can easily be expressed in CCSL by combining the relation *precedence* with a *delay* as in Eq. 8.2.

$$tb_f \boxed{\prec} tb_s \$ n \tag{8.2}$$

8.3 Analysis with VHDL observers

In the presentation of the APB bridge (Section 8.1), we specified two high-level properties about the data transfers through the bridge:

- *P1*: any APB bridge transaction is always as a result of a transaction initiation from the AHB bus. A causality relation expressed as a precedence CCSL constraint: $tb_s \boxed{\prec} tb_f$.
- *P2*: before the current bridge transaction is completed, at most one new request for bridge transaction can be sent by the AHB bus master. This is expressed in CCSL as $tb_f \boxed{\prec} tb_s \$ 2$.

We add a third property concerning the control flow through the bridge.

- *P3*: the APB bridge forbids access when its buffer is full. In CCSL this is represented by a logical clock ($full$) that ticks whenever the buffer gets full. At the RTL level, this results in setting the HREADY signal to low.

These properties are checked against the available VHDL model of a LeonII-based architecture, already mentioned in the introduction. Recall that the observation network directly reflects the abstract syntax of the clock constraint. As a consequence the implementations of the three properties are of increasing complexity. Nevertheless, observation networks for all three properties can be generated systematically. The only manual decision concerns the choice of adapters. In most cases, we simply use adapters that simply produce a pulsed signal when detecting a rising edge or a falling edge. For property P3, we require a more complex adapter to detect the

completion of the write transfer. Indeed, the write transfer completes when the signal PSEL has been asserted HIGH during two consecutive cycles of clock CLK. This implies a sequential behavior.

We can formulate the saturation of buffer (property P3) in terms of logical clocks as in Eq. 8.3. From there, we can derive a CCSL specification (Eq. 8.4).

$$(\forall j \in \mathbb{N}^\star)(\exists k \in \mathbb{N}^\star)(tb_s[j+1] \prec tb_f[j]) \Leftrightarrow (full[k] \equiv tb_s[j+1]) \tag{8.3}$$

$$full \boxed{=} \big((tb_s \$ 1) \wedge td_f\big) - td_f \tag{8.4}$$

The observation network for property P3 is depicted in Figure 6.1. The expression $a \wedge b$, where $\wedge$ denotes the CCSL *inf* operator, defines the slowest clock among all the clocks faster than a and b. The expression $a - b$, where $-$ denotes the CCSL *minus* operator, defines a clock that ticks in coincidence with a whenever b is not coincident with a. For convenience, we can define two auxiliary clocks: $tb_s\_d1 \triangleq tb_s \$ 1$ and $first \triangleq tb_s\_d1 \wedge td_f$, so that Eq. 8.4 can be rewritten as $full \boxed{=} first - td_f$. Figure 8.2 shows an example of execution that respects P3.

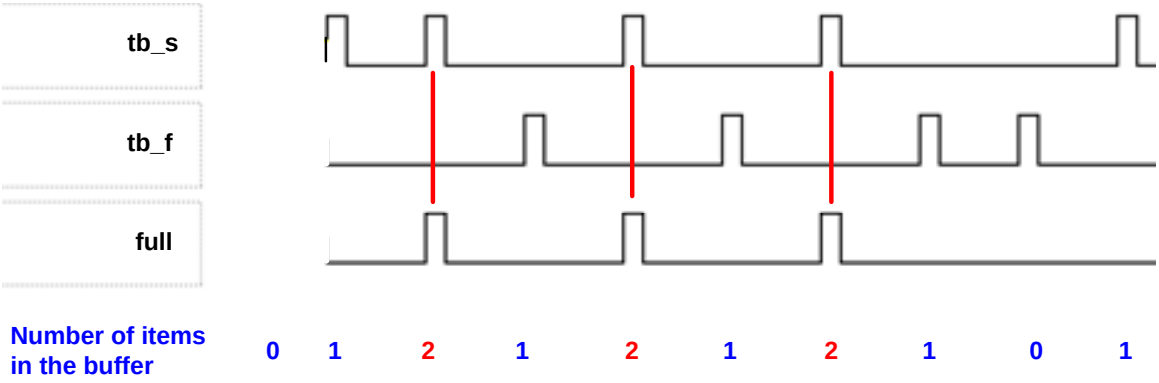

Figure 8.2: Sample Execution of Constraint 3 on CCSL Simulator

The question is now how to relate this observation to the APB bridge behavior. An AMBA slave (*e.g.,* the APB bridge) indicates to its master that it is ready to accept transfers by asserting the signal HREADY. So, when the bridge buffer gets full, the bridge drives signal HREADY to low on the *next* bus cycle. Hence, in the observation network (Fig. 6.1), we had to delay `c_full` for 1 instant of `c_clk`. This is done by

a *defer* generator. Now, since the saturation is manisfested by a low level on signal HREADY, we used a *fallingEdge* adaptor to sense HREADY. `c_full_d1` and `c_invhready` are then observed to be coincident.

The simulation of the LeonII system raises no violation for properties P1 and P2. However, running the simulation against the property observer of P3 has exhibited a problem in the actual implementation of the bridge, that was confirmed as being a weak implementation of the specification. Indeed, the implementation does not support burst transfers.

Part IV

Conclusion

The design process for complex systems makes use of numerous models different in their abstraction level and their nature (underlying model of computation). Usually, the most abstract models are *untimed* or causal. Timing information, regarded as a non-functional property, is introduced in later stages and has the form of "real-time constraints". Since time information may also carry functional intent, some time constraints should be part of the functional models, even at high abstraction levels. We advocate that multiform logical time, as promoted since several years by synchronous languages, is adequate to express a wide variety of time constraints not specific to synchronous systems. Logical time constraints are expressive enough to cover the classical real-time ones but also to represent causal relationships and chronological expressions.

CCSL has been defined as a support to specify multiform logical constraints. It provides a set of classical logical time patterns that can be used to build domain-specific libraries (AADL, East-ADL). CCSL models made out of these libraries provide an explicit timed causality model for other purely syntactic models. Even though CCSL was initially defined as a support to the implementation of the MARTE time model, CCSL can be used with any kind of models, UML-based or not. Clearly, since CCSL only focuses on control aspects and abstracts away values, structure and algorithms, it should be combined with other languages that cover these aspects. Another reason for comparison and combination with other languages is to augment the set of analysis tools that could be used. We have started an effort to increase the interoperability with synchronous languages (Polychrony/Signal, Esterel, Scade) but this effort is ongoing and shall be continued. We have also investigated how to integrate CCSL into the design flows of other kinds of systems, with VHDL or SystemC. This latter aspect raised new problems and in particular justified the need to compare to temporal logics and PSL in particular. Initially, such languages where clearly not in the same scope than CCSL.

Another important aspects for MARTE as a profile is to provide official reference implementations. Papyrus UML has become the reference open-source environment to build MARTE models and therefore a seamless integration with Papyrus is required. This effort has started with the first release of Papyrus and shall continue over the

next year on the brand-new version that should be released at the end of 2010.

As a longer term perspective, we shall continue to promote the use of logical time in models. As the term *logical time* is a bit confusing at first and even though the notion has long been accepted by the embedded, reactive systems community, the model-driven engineering community appears to be less incline to consider timing problems in general and even less logical time. Gaining the acceptance of this community is a challenge for the next years. Indeed, because of the massive spreading of manycore systems, the software engineering community in general and the MDE one in particular can no longer ignore the underlying architectures. It appears more than ever that software must exhibit its concurrency explicitly rather than expecting compilers to extract it automatically so that parallelization becomes possible.

Bibliography

[ABL98] Luca Aceto, Augusto Burgueño, and Kim Guldstrand Larsen. Model checking via reachability testing for timed automata. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer, 1998.

[AMallet09] Charles André and Frédéric Mallet. Specification and verification of time requirements with CCSL and esterel. In Christoph Kirsch and Mahmut Kandemir, editors, *Int. Conf. on Languages Compilers, and Tools for Embedded Systems (LCTES'09)*, volume 44, pages 167–176, Dublin, Ireland, June 2009. ACM SIGPLAN/SIGBED, ACM Digital Library.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoret. Comp. Sci.*, 126(2):183–235, 1994.

[AMD10] Charles André, Frédéric Mallet, and Julien Deantoni. VHDL observers for clock constraint checking. In *Symposium on Industrial Embedded Systems (SIES)*, pages 98–107, Trento, Italy, July 2010. IEEE.

[AMdS07] Charles André, Frédéric Mallet, and Robert de Simone. Modeling time(s). In *10th Intern. Conf on Model Driven Engineering Languages and Systems (MODELS '07)*, number 4735 in LNCS, pages 559–573, Nashville, TN, USA, September 2007. ACM-IEEE, Springer.

[And96] Charles André. Representation and analysis of reactive behavior: a synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29. IEEE-SMC, 1996.

[And04] Charles André. Computing syncharts reactions. *Electronic Notes in Theoretical Computer Science*, 88:3–19, 2004.

[And09] Charles André. Syntax and semantics of the clock constraint specification language (ccsl). Rapport de recherche 6925, INRIA, 05 2009.

[And10] Charles André. Verification of clock constraints: CCSL observers in Esterel. Research Report 7211, INRIA, 02 2010.

[AUT09] AUTOSAR. *Specification of Timing extensions*, November 2009. V1.0.0, R4.0 Rev 1.

[BBKT05] S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis. Testing conformance of real-time applications by automatic generation of observers. *Electronic Notes in Theoretical Computer Science*, 113:23–43, 2005.

[BC88] Gérard Boudol and Ilaria Castellani. Permutation of transitions: An event structure semantics for CCS and SCCS. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 354 of *Lecture Notes in Computer Science*, pages 411–427. Springer, 1988.

[BCC+08] Albert Benveniste, Benoît Caillaud, Luca P. Carloni, Paul Caspi, and Alberto L. Sangiovanni-Vincentelli. Composing heterogeneous reactive systems. *ACM Trans. Embedded Comput. Syst.*, 7(4), 2008.

[BCCSV05] Albert Benveniste, Benoît Caillaud, Luca P. Carloni, and Alberto L. Sangiovanni-Vincentelli. Tag machines. In Wayne Wolf, editor, *EMSOFT*, pages 255–263. ACM, 2005.

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[BDL+06] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *QEST*, pages 125–126, Riverside, California, USA, September 2006. IEEE Computer Society.

[Ber00] G. Berry. The foundations of Esterel. In C. Stirling G. Plotkin and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[BJB05] Belgacem Ben Hedia, Fabrice Jumel, and Jean-Philippe Babau. Formal evaluation of quality of service for data acquisition. In *FDL*, pages 579–589. ECSI, 2005.

[BLMF00] Jean-Michel Bruel, Johan Lilius, Ana M. D. Moreira, and Robert B. France. Defining precise semantics for UML. In Jacques Malenfant, Sabine Moisan, and Ana M. D. Moreira, editors, *ECOOP Workshops*, volume 1964 of *Lecture Notes in Computer Science*, pages 113–122. Springer, 2000.

[BRG+01] J-R. Beauvais, E. Rutten, T. Gautier, R. Houdebine, P. Le Guernic, and Y.-M. Tang. Modeling statecharts and activitycharts as signal equations. *ACM Trans. Softw. Eng. Methodol.*, 10(4):397–451, 2001.

[CCG+07] Philippe Cuenot, DeJiu Chen, Sebastien Gérard, Henrik Lönn, Mark-Oliver Reiser, David Servat, Carl-Johan Sjostedt, Ramin Tavakoli Kolagari, Martin Torngren, and Matthias Weber. Managing complexity of automotive electronics using the East-ADL. In *Proc. of the 12th IEEE Int. Conf. on Engineering Complex Computer Systems (ICECCS'07)*, pages 353–358. IEEE Computer Society, 2007.

[CDE+06] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 180–193. ACM, January 2006.

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In Rajesh Gupta, Yukihiro Nakamura, Alex Orailoglu, and Pai H. Chou, editors, *CODES+ISSS*, pages 19–24. ACM, 2003.

[CPC+04] Dan Chiorean, Mihai Pasca, Adrian Cârcu, Cristian Botiza, and Sorin Moldovan. Ensuring uml models consistency using the ocl environment. *Electr. Notes Theor. Comput. Sci.*, 102:99–110, 2004.

[CRBS08] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. Translating AADL into BIP - application to the verification of real-time systems. In Michel R. V. Chaudron, editor, *MoDELS Workshops*, volume 5421 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2008.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[Esh06] Rik Eshuis. Symbolic model checking of UML activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.

[F02] Stephan Flake and Wolfgang Müller 0003. An ocl extension for real-time constraints. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL*, volume 2263 of *Lecture Notes in Computer Science*, pages 150–171. Springer, 2002.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, CMU, February 2006.

[FGL+08] Marc Frappier, Frédéric Gervais, Régine Laleau, Benoit Fraikin, and Richard St-Denis. Extending statecharts with process algebra operators. *ISSE*, 4(3):285–292, 2008.

[FH07] Peter H. Feiler and Jörgen Hansson. Flow latency analysis with the architecture analysis and design language. Technical Report CMU/SEI-2007-TN-010, CMU, June 2007.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. Use: A uml-based specification environment for validating uml and ocl. *Sci. Comput. Program.*, 69(1–3):27–34, 2007.

[GDB09] Calin Glitia, Philippe Dumont, and Pierre Boulet. Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*, 21(2):105–131, 2009.

[GL00] Wolfgang Grieskamp and Markus Lepper. Using use cases in executable z. In *ICFEM*, pages 111–120, 2000.

[Gun92] Jeremy Gunawardena. Causal automata. *Theor. Comput. Sci.*, 101(2):265–288, 1992.

[Hal92] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[HGGM01] Michael González Harbour, J. J. Gutiérrez García, José C. Palencia Gutiérrez, and J. M. Drake Moyano. Mast: Modeling and analysis suite for real time applications. In *ECRTS*, pages 125–134. IEEE Computer Society, 2001.

[HHJ+05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the symta/s approach. *Computers and Digital Techniques, IEE Proceedings*, 152(2):148–166, March 2005.

[HLR94] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *AMAST '93: Proceedings of the Third International Conference on Methodology and Software Technology*, pages 83–96, London, UK, 1994. Springer-Verlag.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Jan03] Axel Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Systems on silicon. Morgan Kaufmann Publishers, June 2003.

[JHR+07] Erwan Jahier, Nicolas Halbwachs, Pascal Raymond, Xavier Nicollin, and David Lesens. Virtual execution of AADL models via a translation into synchronous programs. In Christoph M. Kirsch and Reinhard Wilhelm, editors, *EMSOFT*, pages 134–143. ACM, 2007.

[JLF08] Rolf Johansson, Henrik Lönn, and Patrick Frey. ATESST timing model. Technical report, ITEA, 2008. Deliverable D2.1.3.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. *Information Processing*, pages 471–475, 1974.

[KK98] Leila Ribeiro Korff and Martin Korff. True concurrency = interleaving concurrency + weak conflict. *Electr. Notes Theor. Comput. Sci.*, 14, 1998.

[KM66] Richard M. Karp and Rayamond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LD06] A. Le Guennec and B. Dion. Bridging UML and safety-critical software development environments. In *Int. Conf. on Embedded and Real-Time Software, ERTS*, 2006.

[Lee00] Edward A. Lee. What's ahead for embedded software? *IEEE Computer*, 33(9):18–26, 2000.

[LM87] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–35, 1987.

[LMdS08] Su-Young Lee, Frédéric Mallet, and Robert de Simone. Dealing with AADL end-to-end flow latency with UML MARTE. In *13th International Conference on Engineering of Complex Computer Systems (ICECCS'08)*, pages 228–233, Belfast, Northern Ireland, apr 2008. IEEE, IEEE Computer Press.

[LS98] E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[MA09] Frédéric Mallet and Charles André. On the semantics of UML/Marte clock constraints. In *Int. Symp. on Object/component/service-oriented Real-time distributed Computing (ISORC'09)*, pages 301–312, Japan, Tokyo, March 2009. IEEE, IEEE Computer Press.

[MAdS07] Frédéric Mallet, Charles André, and Robert de Simone. Modeling of immediate vs. delayed data communications: from AADL to UML Marte. In *FDL*, pages 249–254. ECSI, September 2007.

[MAdS08] Frédéric Mallet, Charles André, and Robert de Simone. CCSL: specifying clock constraints with UML/Marte. *Innovations in Systems and Software Engineering*, 4(3):309–314, 2008.

[McM03] K.L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, July 2003.

[MDAd10] Frédéric Mallet, Julien DeAntoni, Charles André, and Robert de Simone. The clock constraint specification language for building timed causality models. *Innovations in Systems and Software Engineering*, 6(1–2):99–106, March 2010.

[MdS09] Frédéric Mallet and Robert de Simone. *MARTE vs. AADL for Discrete-Event and Discrete-Time Domains*, volume 36 of *LNEE*, chapter 2, pages 27–41. Springer, may 2009.

[MdSR08] Frédéric Mallet, Robert de Simone, and Laurent Rioux. Event-triggered vs. time-triggered communications with UML Marte. In *Forum on Specification, Verification and Design Languages, 2008 (FDL 2008).*, pages 154–159, Stuttgart, Germany, September 2008. IEEE Computer Press.

[Meh10] Aamir Mehmood Khan. *Model-Based Design for On-Chip Systems: using and extending Marte and IP-Xact.* PhD thesis, Université de Nice/Sophia-Antipolis, March 2010.

[Mer74] P. Merlin. *A Study of the Recoverability of Computer Systems.* PhD, University of California, Irvine, 1974.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer, 1980.

[MKMAdS09] A. Mehmood Kahn, F. Mallet, C. André, and R. de Simone. IP-XACT components with abstract time characterization. In *Forum on specification, verification & Design Languages, FDL'09.* ECSI, IEEE Computer Press, September 2009.

[ML02] P.K. Murthy and E.A. Lee. Multidimensional synchronous dataflow. *Signal Processing, IEEE Transactions on*, 50(8):2064–2079, aug 2002.

[MPA08] Frédéric Mallet, Marie-Agnès Peraldi-Frati, and Charles André. Marte CCSL and East-ADL2 timing requirements. Research Report 6781, INRIA, December 2008.

[MPFA09] Frédéric Mallet, Marie-Agnès Peraldi-Frati, and Charles André. Marte CCSL to execute East-ADL timing requirements. In *Int. Symp. on Object/component/service-oriented Real-time distributed Computing (ISORC'09)*, pages 249–253, Japan, Tokyo, March 2009. IEEE, IEEE Computer Press.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

[NPW79] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 266–284. Springer, 1979.

[NPW81] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part i. *Theor. Comput. Sci.*, 13:85–108, 1981.

[OMG05] OMG. *UML Profile for Schedulability, Performance, and Time Specification, v1.1.* Object Management Group, January 2005. formal/05-01-02.

[OMG06] OMG. *Object Constraint Language, v2.0.* Object Management Group, May 2006. formal/2006-05-11.

[OMG08] OMG. *Systems Modeling Language (SysML) Specification, v1.1.* Object Management Group, November 2008. formal/08-11-02.

[OMG09a] OMG. *UML Profile for MARTE, v1.0.* Object Management Group, November 2009. formal/2009-11-02.

[OMG09b] OMG. *UML Superstructure, v2.2*. Object Management Group, February 2009. formal/2009-02-02.

[PEP02] Traian Pop, Petru Eles, and Zebo Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In Jörg Henkel, Xiaobo Sharon Hu, Rajesh Gupta, and Sri Parameswaran, editors, *CODES*, pages 187–192. ACM, 2002.

[Pet87] C.A. Petri. Concurrency theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and their properties*, volume 254 of *Lecture Notes in Computer Science*, pages 4–24. Springer-Verlag, 1987.

[PSL05] IEEE standard for Property Specification Language (PSL), IEEE std 1850-2005, 2005.

[SCK09] Daniel Sinnig, Patrice Chalin, and Ferhat Khendek. Lts semantics for use case models. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 365–370, Honolulu, Hawaii, USA, March 2009. ACM.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In W. A. Hunt Jr. and S. D. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, November 2000.

[Stö05] H. Störrle. Semantics and verification of data flow in UML 2.0 activities. *Electr. Notes Theor. Comput. Sci.*, 127(4):35–52, 2005.

[TCN00] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *International Symposium on Circuits and Systems ISCAS 2000*, volume 4, pages 101–104, Geneva, Switzerland, 2000.

[The04] The East-EEA Project. Definition of language for automotive embedded electronic architecture approach. Technical report, ITEA, 2004. Deliverable D.3.6.

[The08] The ATESST Consortium. East-ADL2 specification. Technical report, ITEA, March 2008. http://www.atesst.org, 2008-03-20.

[Wei08] T. Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design.* The MK/OMG Press, Burlington, MA, USA., 2008.

[Win86] Glynn Winskel. Event structures. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1986.

[Win08] Glynn Winskel. Events, causality and symmetry. In Erol Gelenbe, Samson Abramsky, and Vladimiro Sassone, editors, *BCS Int. Acad. Conf.*, pages 111–127. British Computer Society, 2008.

[YFR08] Lijun Yu, Robert B. France, and Indrakshi Ray. Scenario-based static analysis of uml class models. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 234–248, Toulouse, France, October 2008. Springer.

[YsZ03] Dong Yang and Shen sheng Zhang. Using pi - calculus to formalize UML activity diagram. In *ECBS*, pages 47–54. IEEE Computer Society, 2003.

[YTB+10] Huafeng Yu, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier, Frédéric. Mallet, Charles André, and Robert de Simone. Polychronous analysis of timing constraints in UML MARTE. In *MoBE-RTES 2010*, page 7, Carmona, Spain, May 4th 2010.

[YTG08] Ma Yue, Jean-Pierre Talpin, and Thierry Gautier. Virtual prototyping AADL architectures in a polychronous model of computation. In *MEMOCODE*, pages 139–148. IEEE Computer Society, 2008.

[Zaf05] L. Zaffalon. *Programmation synchrone de systèmes réactifs avec Esterel et les SyncCharts.* Presses Polytechniques et Universitaires Romandes, Lausane (CH), 2005.

Résumé:

CCSL a été construit pour abstraire les données et l'algorithme dans l'intention de focaliser sur les événements et le contrôle. Même si CCSL a été initialement conçu pour servir de modèle de temps au profil UML MARTE, il est devenu un langage de modélisation à part entière dédié à la capture des relations de causalités, chronologiques et temporelles, inhérentes à un modèle. Il est destiné à complémenter des modèles syntaxiques qui eux capturent les structures de données, l'architecture et l'algorithme. Ce document commence par décrire les modèles de parallélisme qui ont inspirés CCSL. Ensuite, le langage CCSL est présenté puis utilisé pour construire des bibliothèques dédiées à deux spécifications standardisées dans les domaines de l'avionique (AADL) et de l'automobile (East-ADL). Finalement, nous introduisons une technique basée sur des observateurs pour vérifier des implantations (Esterel et VHDL) et s'assurer qu'elles respectent bien les propriétés données par une spécification CCSL.

Abstract:

CCSL has arisen from different inspiring models in an attempt to abstract away the data and the algorithms and to focus on events and control. Even though CCSL was initially defined as the time model of the UML profile for MARTE, it has now become a full-fledged domain-specific modeling language for capturing causal, chronological and timed relationships. It is intended to be used as a complement of other syntactic models that capture the data structure, the architecture and the algorithm. This work starts by describing the historical models of concurrency that have inspired the construction of CCSL. Then, CCSL is introduced and used to build libraries dedicated to two emerging standard models from the automotive (East-ADL) and the avionic (AADL) domains. Finally, we discuss an observer-based technique to verify implementations in different languages (Esterel, VHDL) against a CCSL specification.